Connexis User's Guide

Author: Sravan Kumar Lakkimsetti
Mattias Mohlin
IBM

Connexis is a library that can be used for building distributed real-time applications using DevOps Model RealTime. In this document you will learn how to use it and what possibilities it provides.

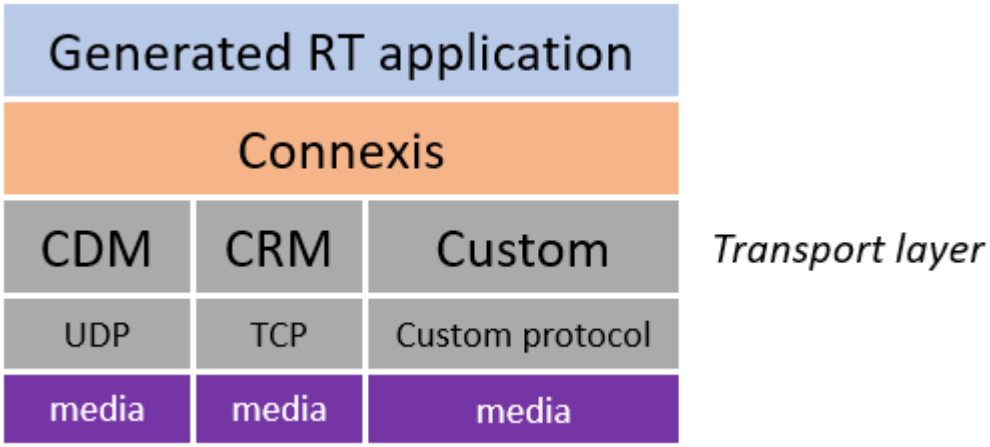
Readers of this document are assumed to have read the document "Modeling Real-Time Applications in Model RealTime" which covers many of the concepts that are explained in more detail in this document.

The document was produced using Model RealTime 10.3. All screen shots were captured on the Windows platform.

Connexis Overview

Applications built with Model RealTime are stateful event-based real-time applications. They link with the TargetRTS (a.k.a the RT Services Library) which contains a communication service that provides both synchronous and asynchronous communication between capsule instances in an application. This service works both when the communicating capsule instances run on the same thread (intra-thread communication) and on different threads (inter-thread communication). However, there is no support in the TargetRTS for the situation when the capsule instances run in different processes, either on the same or on different machines in a network.

This is where Connexis comes into play. Connexis provides an inter-process communication mechanism that allows messages, with or without data, to be sent between capsule instances running in different processes. The transport layer of Connexis is responsible for the actual transmission of messages and their data objects. This layer is built upon current industry-standard technologies, such as UDP and TCP, but you can also use any other custom transport protocol. The picture below shows the architecture of Connexis. CDM means ”Connexis Datagram Messaging” and is built using UDP, while CRM means ”Connexis Reliable Messaging” and is built using TCP.



Connexis Model Library

Connexis is implemented as an Model RealTime model library. To use it you simply import the model library project into your workspace, just like any other model project. The C++ code generated from this model library can be built into a C++ library using ”C++ Library” transformation configurations that are part of that project. However, doing this step is only necessary if you either want to change something in the Connexis implementation (such as implementing a custom transport protocol), or if your application should be built for a platform for which a prebuilt Connexis library does not exist. Model RealTime contains prebuilt Connexis libraries for several platforms (see Supported platforms).

Inter-process communication with Connexis uses unwired ports. These ports are registered with the TargetRTS using a Connexis specific registration string. Other than that the use of Connexis is fully transparent at the model level. This makes it easy to convert an existing monolithic application into a distributed application, simply by replacing a few wired ports by unwired ones. Once these unwired ports have been registered, sending and receiving messages on them is done in the exact same way as in a UML model that does not use Connexis.

Connexis Benefits

The Connexis programming model provides significant value:

- **Built for real-time**
  Automatic mapping of UML communication ports onto a high-performance software backplane.
- **Product-ready, but flexible**
  The software is ready to run as soon as it has been installed but can be adapted to handle application-specific requirements.
- **Simple-to-use programming model**
  Supports client/server type name binding and asynchronous messaging.
- **Support for fault tolerance**
  Detects failures and provides a framework for dealing with faults

Access Transparency

Connexis has a flexible encoding and decoding strategy so that it can work in different types of hardware environments. The endian of the target environment is transparent to the executing application.

Location Transparency

The Connexis Locator service provides location transparency for a distributed application. The application uses service names to refer to the endpoints that are being connected. As a result, the physical address of these endpoints never has to be revealed to the application. Locator features are available for services published on integrated transport addresses. Connexis also supports many different distribution options which allow the design of the application to be very flexible.

The following are the most common types of connections supported:

- **Local connections**

For local connections within the same process, Connexis optimizes communication to be as efficient as directly wired ports. The endpoints of the local connections are registered using service names and Connexis takes care of binding the endpoints together. Once bound, these local connections have the same performance characteristics as two wired ports that have been bound together.

- **Explicit endpoint connections**

Connexis accepts registrations that use explicit endpoint addresses in the registration string. This can be used if the application knows the location of the services that it wants to access. In this way a client can bind to a service using an explicit location and the service name of the desired service.

- **Locator connections**

The Connexis Locator service can be used to find a service (given the service name) anywhere in the distributed application. Once the Locator has been started, one side of the connection registers with it as the publisher, and the other side registers as the subscriber. The Connexis Locator finds the matching endpoints, and feeds them back to the connection service which establishes the connection when both endpoints are available.

Configuration

Connexis supports a wide range of configuration options. This enables the use of Connexis to be very flexible and adaptable to different target environments. For example, configuration options can:

- configure connection audits
- adjust buffer counts and sizes
- adjust thread priorities and stack sizes
- adjust message delivery timing characteristics

The Target Agent and the Connexis Locator Service do not have to be a part of every Connexis application. If your application does not use the Locator service, these components can be left out of the executable's configuration. This helps minimize the size of the Connexis code that gets linked in your executables.

Connexis allows multiple ports to be registered with the same service name. This, coupled with the standard Model RealTime support for port multiplicity, enables several common distribution patterns to be supported by Connexis. Patterns that are not directly supported can usually be implemented very easily in Model RealTime.

The Connexis model library is shipped with full source code. The library can be rebuilt if needed for a custom target configuration (see Porting Connexis to a New Target Configuration).

Fault-tolerance and Reliability

Fault tolerance and reliability are paramount to most real-time systems. Connexis has been designed with these requirements in mind. The following is a list of the different Connexis features that enhance fault tolerance and reliability:

- The Locator Service can be run in simplex or duplex mode. A binary elector is used to determine the health of the primary locator. This provides redundancy and can avoid a "single point of failure" in the distributed application.
- Connexis has been designed for true non-blocking behavior. All potentially blocking system calls are handled by a user-configurable set of helper threads. Name resolving is an example of an operation that makes use of helper threads.
- A heart beat style audit is used over the UDP based connections to detect connection, process or processor failure. This audit is tuneable so that it can be used in a variety of environments. The audit is highly efficient since it monitors user messages to collect status information. This allows explicit "Are you alive?" messages to be avoided. And when explicit "Are you alive?" messages anyway are used, the number of such messages sent per unit of time can be capped to ensure that audits do not overutilize system resources. When a connection-oriented protocol (such as TCP) is used, only a very basic, "Is the connection alive?" protocol is used.
- Buffering policies can be configured between the UML asynchronous messaging controller and the flow-controlled transport message router. This ensures that your model never hangs due to a slow or broken transport connection. When the queue fills up, messages are properly deleted from the system.
- It is easy to build a set of components that meet application specific requirements for fault-tolerance and reliability. The standard generic events, rtBound and rtUnbound, are supported for all Connexis ports, but these notifications can be enabled on a per-port basis and utilized by the application with minimal additional complexity.
- The underlying "try-forever" algorithm can be overridden by the application simply by deregistering the unwired port when certain quality of service parameters are not met.
- Patterns for distribution can be implemented using Connexis as the underlying distribution mechanism. For example, a single publisher can actually hide multiple distributed connections to a replicated service.
- Resource constraints can be placed on publishers of services to limit the number of subscribers per publisher.
- The allocation of unwired ports to capsules is under the control of the designer and since the deployment of capsules onto threads also is under control of the application, the application can dynamically control which resources are being used.
- Services can be ranked according to the preferred order of use. This ranking is done with full, dynamically updated knowledge of the resource limits. If a given service with a certain rank is full, one of a lower rank will be used, if available.

- Configurable system audits verify that the internal system connection state matches across the entire system.

Supported Platforms

Model RealTime contains prebuilt Connexis libraries for the following platforms:

- LinuxT.x64-gcc-12.x (Linux 64 bit with gcc 12.x)
- WinT.x64-MinGw-12.2.0 (Windows 64 bit with MinGW 12.2.0)
- WinT.x86-VisualC++-17.0 (Windows 32 bit with Visual Studio 2022)
- WinT.x64-VisualC++-17.0 (Windows 64 bit with Visual Studio 2022)
- WinT.x64-Clang-16.x (Windows 64 bit with Clang 16.x)

To build Connexis for another platform, please follow the instructions provided in the chapter Porting Connexis to a New Target Configuration.

Connexis Terminology and Definitions

The table below summarizes some of the terms and abbreviations used throughout this document.

Term	Definition
CDM	Connexis Datagram Messaging. A thin layer on top of UDP that provides additional support for connection auditing and quality of service parameters.
CRM	Connexis Reliable Messaging. A thin layer on top of TCP that provides additional support for connection auditing and quality of service parameters.
DCS	Distributed Connection Service. This is the key component used for connecting and managing the different parties in a connection. This is also the name of the Connexis model library.
DNS	Domain Name System (or Service). An Internet service that translates domain names into IP addresses. Because domain names are alphanumeric, they are easier to remember; however, the Internet is really based on IP addresses. Every time you use a domain name, a DNS service must translate the name into the corresponding IP address. For example, the domain name www.example.com might translate to 198.105.232.4.
duplex locator service	Refers to a configuration in which there are two locators. In normal operation, one locator acts as the active locator (primary) and the other acts as the standby locator (backup). This configuration is used to prevent a locator from being a single point of failure.
endpoint	An endpoint is an explicit network address used for finding a peer object. It consists of a protocol followed by a colon, followed by a protocol-address. For example: `cdm://ipaddress:port`. The endpoint is realized as an executable that runs on the specified network host, and that listens for messages on the specified port.
ILS	Internal Layer Service. This is the connection service that is built into the

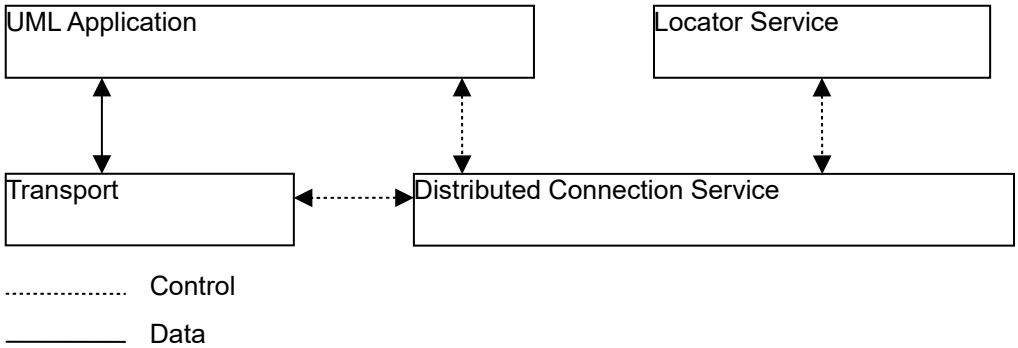
	TargetRTS of Model RealTime. It can only be used for establishing intra-process connections between ports.
IPC	Inter-Process Communication. This is a broad term that is being used to describe any mechanism for sharing information between processes. This could be something as simple as shared memory or something as sophisticated as CORBA.
Locator	The Connexis Locator Service is a configurable service that is used to look up the physical location of an object given a service name for that object.
notification	The term used to describe the process of sending a message to a capsule instance to inform it when one of its ports has been connected to (rtBound), or disconnected from (rtUnbound), its peer.
SAP	Service Access Point. Term used to describe an unwired port that is participating in a connection as the subscriber (on a client capsule).
simplex locator service	Refers to a configuration in which there is only one locator. This configuration does not provide redundancy. See also duplex locator service.
SPP	Service Provisioning Point. Term used to describe an unwired port that is participating in a connection as the publisher (on a server capsule).
TargetRTS	Target Run-time System. Also known as the RT Services Library. A C++ library that implements all UML-RT services needed by an application that is generated from Model RealTime.
TCP/IP	Transmission Control Protocol / Internet Protocol. A connection-based transport protocol.
Transport	The underlying protocol that is being used to pass messages and data between communicating objects.
Transport Integration Framework (TIF)	A framework that lets 3rd parties (including developers) add additional transports for use in Connexis.
UDP	User Datagram Protocol. A connection-less transport protocol.
UML	Unified Modeling Language. An industry-standard modeling language used to model object-oriented software systems.
UML-RT	UML RealTime. An extended subset of UML which provides useful services for building stateful event-based real-time application. Applications are designed in Model RealTime (with or without Connexis) to model the software that is being built.
unwired port	An unwired port is a UML object that can have connections specified using registration strings. These strings can be specified at either design time or run-

	time. An unwired port is either an SPP port or an SAP port, and at run-time two ports of those different kinds can be dynamically connected to enable message and data exchange.
virtual circuit	A virtual circuit is the term used to refer to a connection that has been established between two endpoints. This refers to a connection between a single subscriber (SAP port) and a single publisher (SPP port). Connexis has a static (but configurable) limit on the maximum number of virtual circuits across process boundaries.
wired port	A wired port is a UML object that can have an explicit connection (to another wired port) specified at design time. The connection will be established at system initialization time, or at run-time if the port is contained in a capsule that is optional or plug-in.

Connexis Application Layers

Connexis allows multiple Model RealTime-generated executables to be connected in a robust and reliable manner. Executables are networked by connecting unwired ports across process boundaries.

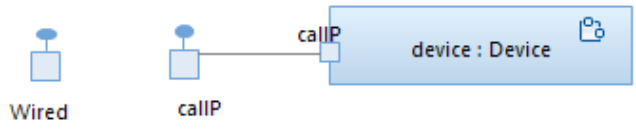
An Model RealTime application that uses Connexis to implement its inter-process communication has the high-level architecture shown below. The control paths that are shown, indicate the components that are involved in registering and deregistering endpoints in the UML application. All data that is sent between endpoints in a Connexis-enabled application goes through the Transport component.



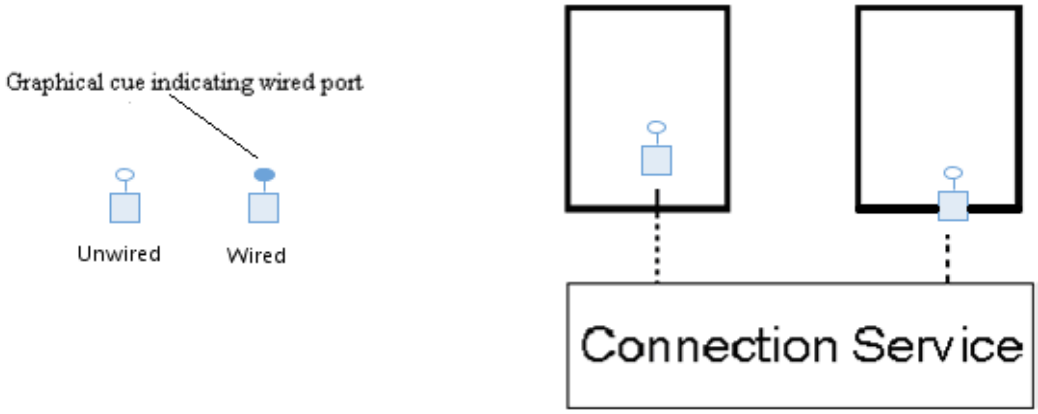
Let's have a look at these different application layers.

UML Application

In the UML application ports are used to send messages between the capsules in your model. There are different kinds of ports. The most common type of port is a wired port. Wired ports are visibly connected to other wired ports by means of connectors. Wired ports are represented graphically with two connected squares in the oval part of the port icon.



Another type of port is the unwired port. Unwired ports are the primary method for establishing Connexis connections. Once you have created an unwired port, you can specify the connection service, protocol, and endpoint address that it will use by registering the port with the TargetRTS. This registration can either be done automatically (design-time) or through application code (run-time). The Connection Service shown in the picture below is the Distributed Connection Service of Connexis, i.e. the DCS is one implementation of a Connection Service.

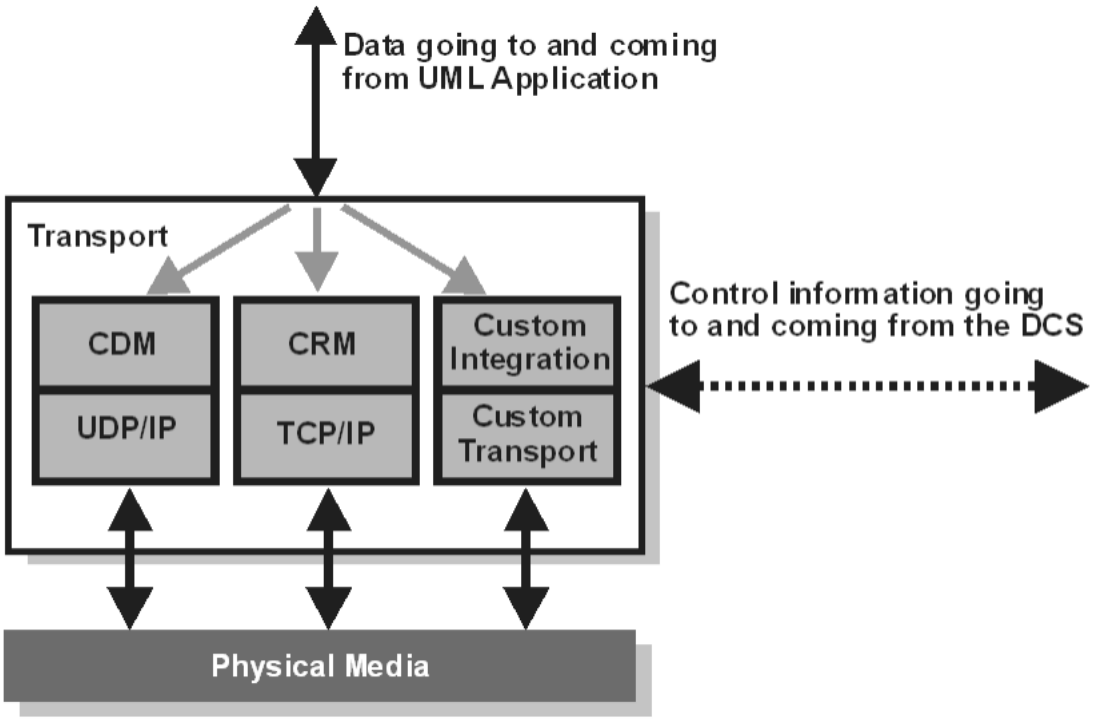


Distributed Connection Service

The Distributed Connection Service (DCS) is the connection service that is provided with Connexis. It is responsible for maintaining information about the unwired ports that have been registered with it. The DCS is part of the system that is responsible for establishing connections between unwired ports. It does this by parsing the registration strings that are passed in when an unwired port is registered.

Transport

The Transport is the component that is responsible for sending and receiving data between processes. It manages any incoming or outgoing data buffers and encodes and decodes data. A more detailed break-down of the Transport component is shown below.



Locator

Another key component of a robust distributed system is a fault-tolerant name service. A name service is used to find the actual location of a service given a predetermined service name. A well-known example of a name service is the Domain Naming Service (DNS) that is widely used on the Internet. The principle function of a name service is to look up a specific address when it is given a service name. This isolates the calling application from changes in the physical addressing of network components. In Connexis, the Locator Service provides the name service functionality.

The Locator Service actually does a bit more than just operating as a name server. The Locator can be configured to arbitrate between more than one endpoint that provides the same service and it can also be set up to run in duplex mode, which allows a backup Locator to automatically take over when the primary Locator fails.

An endpoint is defined to be the combination of a transport protocol and the address of a specific port of an object in a distributed application. For example, `cdm://address:port`. If an explicit endpoint is provided, then the Locator service is not needed and the client will try to directly connect to the server at the specified endpoint location. However, if an explicit endpoint is not provided then the Locator is contacted. The Locator returns an endpoint that is then used by Connexis to choose the appropriate service provider or peer. The service name by which an endpoint is referred, is specified as part of the registration string of that endpoint.

The Connexis Locator Service supports both a primary and a backup locator. In this way, a distributed application can be made more robust by ensuring that the name server does not become a single point of failure.

The Connexis HelloWorld Sample Model

Model RealTime contains a sample Connexis model which demonstrates the use of the Connexis Locator Service and how it can be used to provide a backup locator service in a distributed environment. Open the sample model by performing **File – New – Example**. Under the ”UML Capsule Development” category, select the ”Connexis_HelloWorldOverflowToBackupService” example.

Import the Connexis library model to your workspace (**File – Import – General – Existing Projects into Workspace**). The project to import is located in the Model RealTime installation under `rsa_rt/Connexis/DCS`.

The Connexis HelloWorld sample model contains two servers, a client, and the Connexis Locator service, each running as independent applications. The servers speak different languages (either English or French). Initially, a client is bound to the server that comes up first. Once bound, the client makes periodic requests to the server and the server sends back a greeting in the language it speaks.

If the server to which the client is bound becomes unavailable, the client is notified of the connection loss. It then rebinds to the backup server, which starts responding to the client requests in the language it speaks. For example, if the client is initially bound to the English server, it will start receiving greetings in English. If you then terminate the English server, the connection will be lost momentarily. The client will then be rebound to the French server, and after a short while the client will start receiving the greetings in French instead.

```
Hello World!!!
Hello World!!!
Hello World!!!
Hello World!!!
Lost connection to server
cancel(application(Client)<TOP>.timer): Invalid timer id.
Salut le monde!!!
Salut le monde!!!
Salut le monde!!!
Salut le monde!!!
Salut le monde!!!
```

Another thing demonstrated by this sample is overflow handling. The English server is given a higher rank so that it acts as the primary server. The clients connect to the primary server until the primary server has reached its full capacity (2 clients in the example). Any subsequent client that is launched will be connected to the French backup server instead.

Building and Running the HelloWorld Model

Build the four executables using the TCs that are located in a folder named after your platform. For example, for MinGW the folder is called ”TCs-win64-MinGW”. Note that the TCs for different platforms have the same names (it’s only the folder name that is different). It is therefore recommended to have the ”Transformation Configuration Files” filter in the Project Explorer turned off so you can see the *.tcjs files under the folders and not just under the Transformation Configuration virtual folder.

Go to the target folder for each TC and start the generated application: Start the locator application first:

```
Locator.EXE -CNXep=10000 -CNXlp -CNXui=PrimaryLocator
-CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

Then start the English server:

```
EnglishServer.EXE -CNXep=10020 -CNXui=EnglishServer -CNXlpep=cdm://localhost:10000
-CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

And finally start the French server:

```
FrenchServer.EXE -CNXep=10030 -CNXui=FrenchServer -CNXlpep=cdm://localhost:10000 -
CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

Now start four independent clients, making sure to give each one a unique name and port:

```
Client.EXE -CNXep=10010 -CNXui=Client1 -CNXlpep=cdm://localhost:10000 -
CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

```
Client.EXE -CNXep=10011 -CNXui=Client2 -CNXlpep=cdm://localhost:10000 -
CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

```
Client.EXE -CNXep=10012 -CNXui=Client3 -CNXlpep=cdm://localhost:10000 -
CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

```
Client.EXE -CNXep=10013 -CNXui=Client4 -CNXlpep=cdm://localhost:10000 -
CNXlbep=cdm://localhost:10001 -URTS_DEBUG=quit
```

You should see that client 1 and 2 start to print "Hello World!!!" since they were connected to the English server, while client 3 and 4 print "Salut le monde!!!" since they were connected to the French server (because the specified capacity of the English server only allows two connected clients).

You should also see that if you terminate the English server, client 1 and 2 after a while get reconnected to the French server and therefore start to print the message in French instead.

Using Connexis

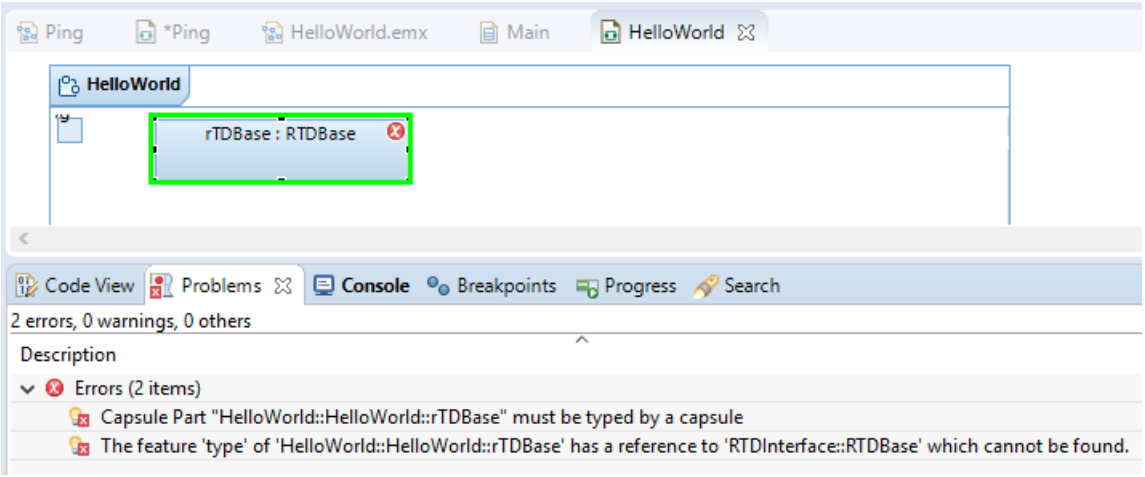
The steps for using Connexis in a model are:

1. Import the Connexis library model to your workspace (**File – Import – General – Existing Projects into Workspace**). The project to import is located in the Model RealTime installation under `rsa_rt/Connexis/DCS`.
2. Perform one of the steps below
   a) If you use Connexis on one of the platforms for which a prebuilt library is available (see Supported Platforms), and you have not customized the Connexis model library, then add an external library TC from the Connexis model as a prerequisite of your application TC. The names of these external library TCs are `prebuilt_<platform>.tcjs`.
   b) If you have customized the Connexis library model, or are using a platform that is not supported out of the box, then use instead the library TC that builds the Connexis library. Add it as a prerequisite to your application, so that the Connexis library gets built automatically when you build your TC. The names of these library TCs are `<platform>.tcjs` and you can of course create your own such library TCs to target other platforms. See Create a C++ Library TC for Connexis for more information.
3. Apply the Connexis profile to your top-level package. This is not a mandatory step, but the Connexis profile can help you create the necessary model elements by automatically updating your model when you apply Connexis-related stereotypes.
4. Create your Connexis application model using regular UML-RT elements as well as elements from the Connexis library (either added manually or by means of using the Connexis profile). More information about this can be found in the chapter Connexis Services.

In addition to these steps, there are also general design rules that must be followed to ensure that the Connexis components have been initialized properly before they are used.

Migrating Old Connexis Models

Connexis models created in older versions of Model RealTime, and Connexis models imported from IBM Rational Rose RealTime (Rose-RT), use an old version of the Connexis profile that contained a few duplicated elements in the RTDInterface package. These duplicates have now been removed, and your model should instead use the corresponding elements from the Connexis library model ("DCS"). You can find the references that have become broken by performing the **Validate** command on your model. Here is an example of such a broken reference:



You can fix a broken reference by right-clicking on the problem (the second error shown in the picture above), and perform the **Quick Fix** command. Then select **Search or browse for a valid reference**. You can then browse to the Connexis library model ("DCS") and locate the valid target for the reference in the package DCS::Logical View::DCSModelInterfaces::RTDInterface.

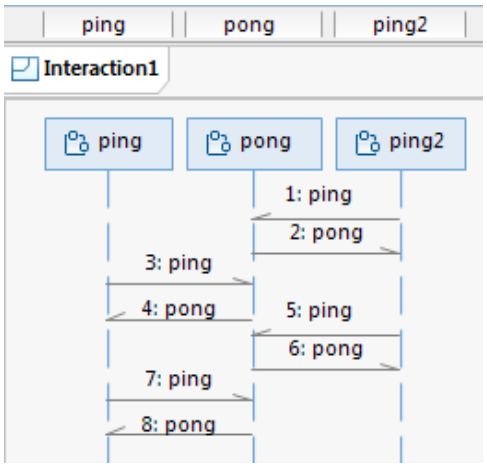
Note that it is no longer mandatory to use the Connexis profile. However, if your model already has it applied it is recommended not to remove it. If you attempt to remove the profile, parts of your model may be implicitly removed too, and you then have to recreate those parts which could be tedious. There is no harm in keeping the Connexis profile applied even if it is no longer mandatory to use.

Tutorial: Building a Distributed Application with Connexis

The best way to learn how to use Connexis is to build a small distributed application that uses it. This tutorial will take you through the steps that are required to create, build and execute a Connexis-enabled application. The only prerequisite for following this tutorial is that you have Model RealTime with Connexis installed (note that Connexis is an optional installation component that is not installed by default).

Overview

The application to be created in this tutorial is a simple “ping pong” application. Each client sends a "ping" message to the server, and the server responds with a "pong" message. The sequence diagram below shows the messages that are sent between the Ping (client) capsule instances and the Pong (server) capsule instance.



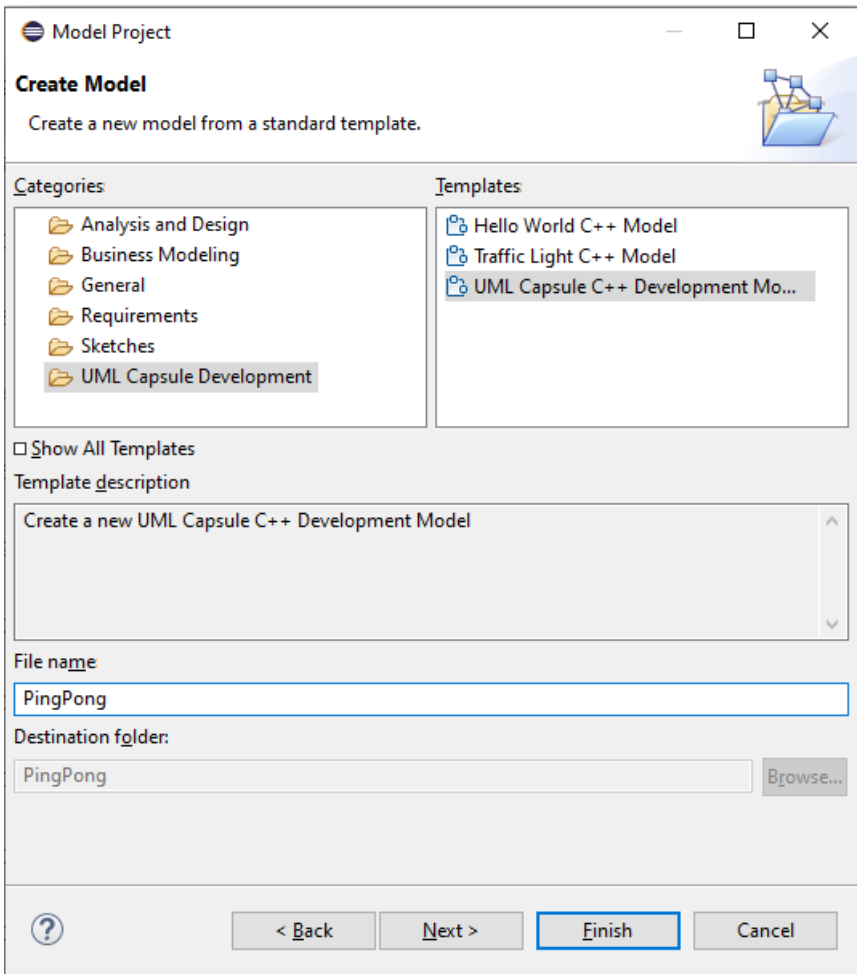
Registration is accomplished using the Locator Service. The necessary command line options for starting the applications are also presented.

This application is created in two iterations:

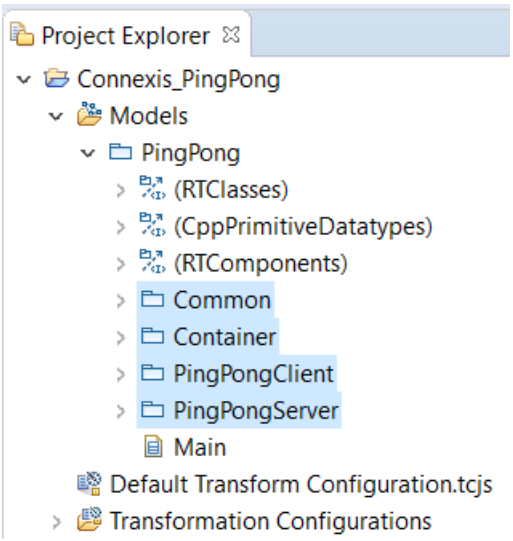
- **Iteration 1: Creating the Application Model**
  Creates the basic architecture using wired ports to connect the Ping and Pong capsules. This also involves a third capsule which acts as the container for the Ping and Pong capsules.
- **Iteration 2: Use Connexis to Make the Application Distributed**
  Makes modifications in the application so that Ping and Pong communicate through unwired ports that make use of Connexis connections.

Iteration 1: Creating the Application Model

1. Start by creating a new model project (**File - New - Model Project**). Give the new project the name "Connexis_PingPong".



2. Find the created project in the Project Explorer, and select the top PingPong package. Use the context menu command **Add UML - Package** to create four packages under the PingPong package.



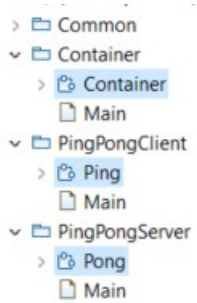
Using packages for organizing the elements of an application may not be necessary for a small and trivial application like this one, but for larger applications it can help to group related elements into packages.

3. Select the PingPongClient package and create a Ping capsule in it. (**Add UML - Capsule** in the context menu)
4. Create a Pong capsule in the PingPongServer package.

In this application, the Ping capsule plays the role of the client. It is referred to as the client because it only manages a single connection. The server side of the application is responsible for managing multiple connections (one for each client it is connected to). The Pong capsule plays the role of the server in this application. Note that in a real-world distributed application it is not uncommon that the same object both can play the role of client and server, for different communication scenarios. However, in this simple application Ping capsule instances are always clients, and a single Pong capsule instance is the server.

5. Create a Container capsule in the Container package.

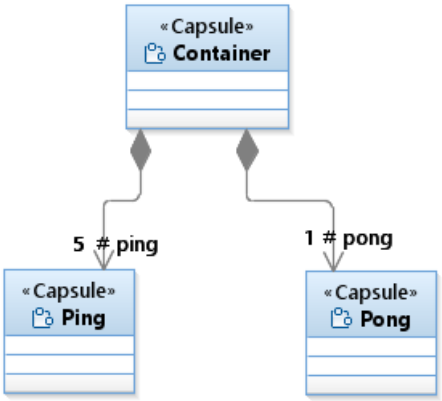
You now have all three capsules needed for this application:



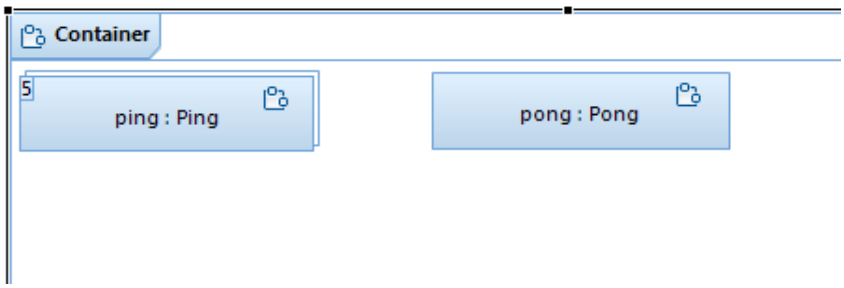
6. Now it's time to define the communication between the Ping and Pong capsules, by creating a protocol. Select the Common package and use the context menu command **Add UML - Protocol** to create a protocol called "PingPong".
7. Create two events in the protocol; an In Event called "pong" and an Out Event called "ping".

The convention we use here is to define the protocol from the perspective of the client. That's why "pong" is an In Event, because it will be received by the client, and "ping" is an Out Event because it will be sent by the client. Note that this way of defining a protocol is just a convention. It also works to define a protocol from the perspective of the server. The most important is that you are consistent throughout the whole application, to make the model easier to understand. However, a benefit with defining protocols from the perspective of the client is that you then only need to conjugate one port (the one in the server), while all client ports can remain non-conjugated. There are usually more client ports than server ports, which is why this convention means less modeling effort.

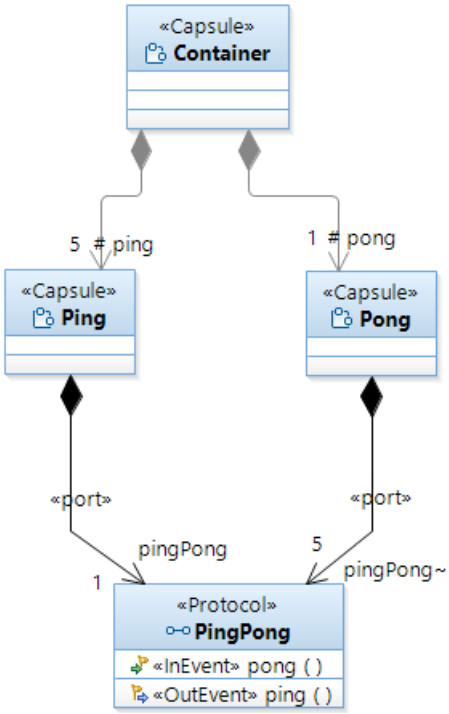
8. Rename the Main class diagram in the top PingPong package to "Architecture". Drag and drop all three capsules onto the class diagram and create composition associations from Container to Ping and Pong. By doing so you add two capsule parts ("ping" and "pong") in the Container.
9. Set the multiplicity of the "ping" capsule part to 5. Keep the "pong" capsule part multiplicity unchanged. This means we will have 5 Ping clients and 1 Pong server in the application.



If you open the structure diagram of the Container capsule, you can see the capsule parts there too.



10. Create a port in both the Ping and Pong capsule. This can be done in many ways, for example by dragging the PingPong protocol onto the Architecture class diagram and then creating composition associations from Ping and Pong to the PingPong protocol. Keep the default port name that is derived from the name of the protocol ("pingPong").
11. Set the multiplicity of the port in the Pong capsule to 5 so that it can connect to 5 different Ping clients. Also, using the Properties view, mark that port as conjugated. This swaps the meaning of In Events and Out Events in the protocol so that Pong can receive the "ping" Out Event and send the "pong" In Event.



12. Create a port "log" in both the Ping and Pong capsules. Type this port with the predefined Log protocol. The Ping client will use this port to write a log message when it receives the "pong" event from the server, and the Pong server will write a log message when it receives the "ping" event from a client. Note that contrary to the "pingPong" port, these ports should not be service ports, so unmark the **Service** checkbox in the Properties view. A non-service port is not part of the communication interface of a capsule, and can only be used by the capsule itself.
13. To let the Ping and Pong capsules communicate we need to connect the "pingPong" ports. For now we will connect them with a connector, but in the next chapter (Iteration 2: Use Connexis to make the application distributed) we will instead make the ports non-wired and use Connexis for establishing the communication path. Create the connector in the structure diagram of the Container capsule.

Now it's time to implement the behavior of the Ping and Pong capsules. We use state machines for this.

14. Select the Ping capsule, either in the "Architecture" class diagram or in the structure diagram of the Container capsule. Perform the command **Open State Machine Diagram** from the context menu to open the state machine of the Ping capsule. It has a default simple state machine just consisting of one state and an initial transition.
15. Rename the state to "ready".
16. Select the initial transition and use the Code view to enter the following effect code: `pingPong.ping().send();`

    This means that as soon as an instance of the Ping capsule is created, and its state machine starts to execute, it will send the "ping" event on the "pingPong" port.

17. Use the context menu command **Add UML - Transition** to create an internal transition for the "ready" state.
18. Use the Triggers page in the Properties view to create a trigger for this transition. The transition should trigger when the "pong" event is received on the "pingPong" port. Also rename the transition to "pong". To make the state machine readable it can often be good to name the transition according to the event that triggers it, at least when it can only be triggered by one event as in this case.
19. Use the Code view and enter the following effect code for the "pong" transition:
```
log.log("received a pong");
log.commit();
pingPong.ping().send();
```

The Ping state machine should now look like this:

Why did we create "pong" as an internal transition, instead of drawing an external transition line from the "ready" state and back to itself? Because we want the state machine to stay in the "ready" state when it handles the "pong" event. This both makes the state machines easier to understand, the state chart diagram easier to draw, and also the generated application to run slightly faster (no need to first exit the state and then enter it again).

20. Now repeat the steps above for creating the state machine of the Pong capsule. The only difference is that the internal transition now should be called "ping" and trigger on the "ping" event. The initial transition should be left empty (since there is no need for the server to do anything when it starts up), and for the "ping" transition you should enter the following effect code:
```
log.log("received a ping");
log.commit();
rtport->pong().reply();
```

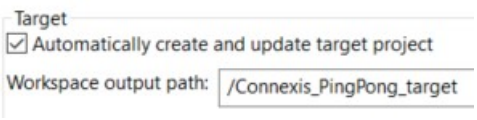
Here we have used "rtport" which for each transition effect code refers to the port on which the event that triggered the transition was received. "rtport" is mainly useful when you want to reply on a received event on the same port. In that case you don't have to refer to the port by its name. Feel free to instead send the event in the same way as was done in the Ping state machine if you think it makes the code more readable.

Build and Run the Model

To build and run the model you have created, you need a transformation configuration (TC). This is a file that contains all build settings and other properties that control how the model is transformed into an executable application. Your model project contains a default transformation configuration called "Default Transformation Configuration". It contains many useful default settings, but some updates are needed before it can be used for building your application.

Double-click on "Default Transformation Configuration" to open the TC editor. Set the following TC properties:

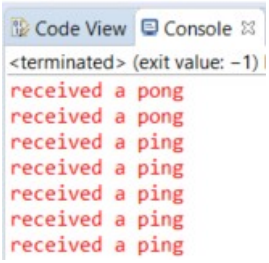
- On the Main tab, click "Automatically create and update target project" and specify a workspace path for where to create the CDT project with generated C++ files. For example:



- On the Code Generation tab, set the "Top capsule" property to the Container capsule. This specifies the capsule that will be incarnated when the application starts up.
- On the Target Configuration tab, set the "TargetRTS configuration" property to match your target platform (OS and compiler). Also specify the make dialect using the "Make type" property.

Save the TC and build it by using the context menu command **Build**. C++ code will be generated, and compiled and linked into an executable. The executable is placed in the "default" folder of the specified target project, and is called "executable".

Run the application by using the TC context menu command **Run As - RealTime Application**. After a short while you should see the log message printouts in the Console view:



These printouts show that your ping pong application works correctly. Each of the 5 clients sends a "ping" and the server replies with a "pong" to each of them. Now it's time to convert this monolithic application into a distributed application using Connexis.

Iteration 2: Use Connexis to Make the Application Distributed

Before converting the sample PingPong model built in Iteration 1 to use Connexis, you may want to copy the project and paste it as a "PingPong_Iteration1" project. Thereby you can compare the models for the distributed and monolithic PingPong applications later.

The first thing we need to do, in order to make our application use Connexis, is to replace the wired ports with non-wired ports. Connexis manages connections that are established between unwired ports. Unwired ports are ports whose connections are defined at run-time. Unlike wired ports, unwired ports cannot be connected by connectors. The connections established between unwired ports can also be removed and reestablished at run-time. Hence, unwired ports allow for more dynamic connections in an application. Unwired ports can be used also in non-Connexis applications when you need more dynamic connections in an application. However, only use them when necessary since the main drawback with non-wired ports is that you no longer can see the connection paths graphically in composite structure diagrams. This could make an application harder to understand.

1. Open the structure diagram of the Container capsule and delete the connector by right-clicking on it and performing the **Delete from Model** command.
2. Convert the "pingPong" ports of the Ping and Pong capsule to become unwired ports. You can do it from the Properties view by unmarking the "Wired" checkbox.
3. Mark the "Publish" property for Pong's port. This makes it a Service Provision Point (SPP), i.e. a capsule port for a server that publishes a service through the port. Also set the "Registration Kind" property to "Application". This means that the TargetRTS will not automatically attempt to register the port to establish connections at run-time. Instead we will perform the registration programmatically using transition code.

4. For Ping's port we leave the "Publish" property unset. This makes it a Service Access Point (SAP), i.e. a capsule port for a client that accesses a service through the port. Also for this port we set the "Registration Kind" property to "Application" to let us manage the registration of the port from code.

When a matching SAP and SPP port have been registered, Connexis can establish a dynamic connection between the ports. Our responsibility as application developers is not to establish the actual connection, but only to register a port when its owning capsule is ready to start communicating. This simplifies our code significantly, since we don't need to handle the actual work of setting up (or removing) connections, which in case of a distributed application can be a non-trivial task.

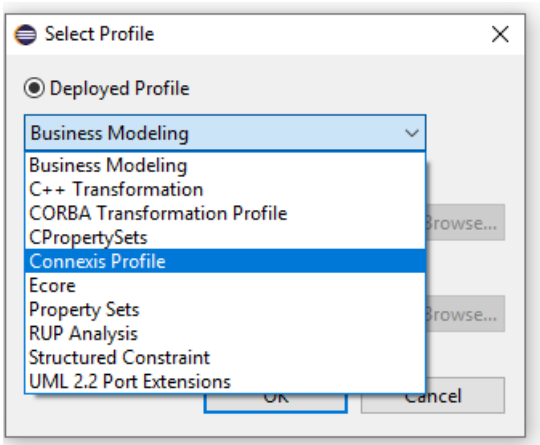
5. For Ping's port we also need to set the "Notification" property. This is necessary so that the Ping capsule will be notified (by means of the rtBound event) when its port get connected to Pong's port at run-time.

It is common to only have the "Notification" property turned on for SAP ports on clients, and not on SPP ports on servers. This is because a server usually just waits for incoming events from clients, and replies to them when they arrive. A client, on the other hand, needs to know when the server is available, since it cannot make the server request until then. However, in case a server needs to somehow keep track of the connected clients, it too may benefit from setting the "Nofication" property.

Up until now we have only used standard UML-RT concepts to create our model. Now it's time to start using Connexis.

6. Import the Connexis model library to your workspace (**File – Import – Existing Projects into Workspace**). The project to import is located in the Model RealTime installation under `rsa_rt/Connexis/DCS`.

7. Select the top-level PingPong package. Use the Profiles tab in the Properties view to apply the Connexis profile. You find it in the Deployed Profiles drop down.



Using the Connexis profile is not mandatory but it makes it easier to add the necessary Connexis elements to your model. When you apply stereotypes from the Connexis profile, needed Connexis elements will be automatically added to your model.

8. Select the Ping capsule. Use the Stereotypes tab in the Properties view to apply the stereotype "Connexis Feature" to the capsule.

When this stereotype is applied to the capsule, a capsule part is automatically added to it. It is typed by RTDBase from the Connexis library. This capsule part exposes various Connexis services for the capsule.

9. Set the "RTD InitStatus Port" property of the "Connexis Capsule" stereotype to True.

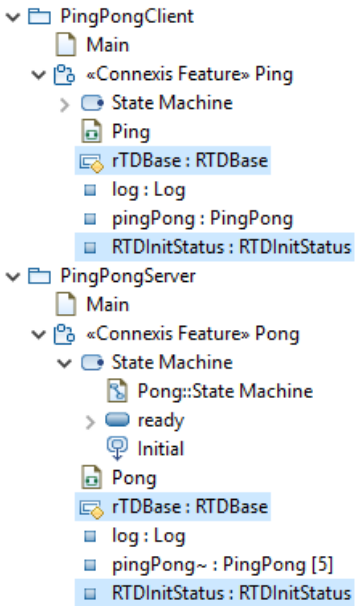
Stereotype Properties:

Property	Value
Connexis Capsule	
CDM Transport	False
CRM Transport	False
Locator Functionality	False
RTD InitStatus Port	True
RTD Metrics Port	False
Target Agent Functionality	False

When this stereotype property is set to True, an RTDInitStatus port is automatically added to the capsule. The purpose of this port is to notify the capsule when the Connexis library has been properly initialized and is ready to be used. It is an SAP port that at run-time will be connected to the corresponding SPP port on the RTDBase capsule. The port's Notification property is enabled which means the capsule can wait for the rtBound event to arrive on the port. Before it arrives Connexis is not ready to be used, and any attempt to use it will fail.

10. Now repeat the same steps for the Pong capsule. First apply the "Connexis Feature" stereotype and then set the "RTD InitStatus Port" property to True.

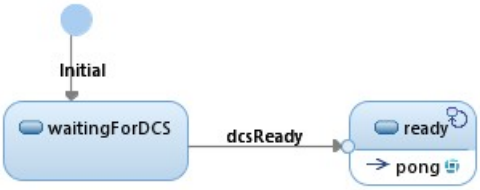
Confirm in the Project Explorer that the Ping and Pong capsules now look as shown below (elements that were added automatically by the Connexis profile have been selected):



As you may have already guessed, the next step is to update the state machines of Ping and Pong to wait for the rtBound event on the RTDInitStatus ports.

11. Open the Ping state machine and add a new state "waitingForDCS". The capsule will wait in that state until Connexis has been initialized. Delete the old initial transition and create a new one that instead targets the "waitingForDCS" state. Then add a transition "dcsReady" from the "waitingForDCS" state to the "ready" state. Finally add a trigger for the rtBound event on the RTDInitStatus port for this transition.

The Ping state machine should now look like this:



12. Repeat the same procedure for the Pong state machine.

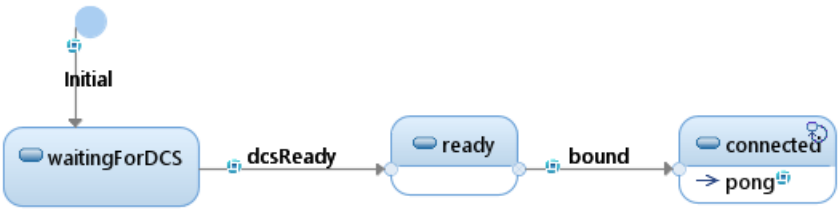
Once Connexis has been initialized, it can establish a connection between the pingPong ports of Ping and Pong. Remember that these ports now are unwired and with the "Registration Kind" property set to "Application". This means they are no longer connected immediately when the capsule instance is created. Instead, they will be connected programmatically, and this can only be done when Connexis has been initialized.

13. Open the Ping state machine and add a new state "connected". Add a transition between the “ready” state and the “connected” state and call this transition “bound". Add a trigger for the transition so it triggers when rtBound arrives on the pingPong port.
14. Select the "bound" transition and use the Code view to enter the following effect code: `pingPong.ping().send();`

This was the code we previously had in the initial transition. Note that the introduction of Connexis caused us to delay the sending of this event until we are sure that Connexis has been initialized and the pingPong port has been bound to the Pong server.

15. Move the "pong" transition from the "ready" state to the "connected" state. Unfortunately it is not possible to simply move this transition from one state to another in a diagram nor in the Project Explorer (a future version of Model RealTime will hopefully support this). Instead you have to temporarily make the transition external (in the Properties view), then move each transition line endpoint from the "ready" to the "connected" state, and finally make the transition internal again.

The Ping state machine should now look like this:



The connection of unwired ports in your model uses the registration string that has been specified for the ports (using the "Registration Override" property). If no registration string is specified, it defaults to the name of the port. Without Connexis the ports are always connected within the same TargetRTS. In this case you can use any registration string as long as the same string is used for SAP and SPP ports. However, when Connexis is used there are three different ways how the lookup of a registered port can be done:

- **Locally**
  This is the same as if Connexis is not used, i.e. a registered target port is looked for in the current TargetRTS.
- **Explicit address**
  Here the registration string contains an explicit address of where the application with the target port is located. Connexis will only look for the port in that specific location.

- **Locator**
  Here the Locator service is used for finding the location where to look for the target port.

The format of the registration string determines which of these three ways that will be used. For our PingPong application we will use the Locator, where the registration string has the following syntax:

```
dcs:/pingpong
```

The name "pingpong" could of course be changed to anything, as long as the SAP and SPP ports agree on the same name. It should also be noted that Connexis supports certain parameters to be specified in the registration string, for example to set which transport protocol that should be used, but it is beyond the scope of this tutorial to go into those details.

Now let's add the code for registering the pingPong ports.

16. Open the Ping state machine. Select the "dcsReady" transition and type the following effect code in the Code view:
```
if (!pingPong.registerSAP("dcs:/pingpong")) {
  log.log("Ping registration failed");
  log.commit();
}
```
17. Open the Pong state machine. Select the "dcsReady" transition and type the following effect code in the Code view:
```
if (!pingPong.registerSPP("dcs:/pingpong")) {
  log.log("Pong registration failed");
  log.commit();
}
```

As the final step, before our model is ready to be built, we must specify a couple of more configuration settings for Connexis. We need to decide where the Locator service should be placed. Ping and Pong will run in their own executables and any of these could contain the Locator service. We choose to place it in the server application, i.e. in Pong.

18. Select the Pong capsule and go to the Stereotypes tab in the Properties view. Set the "Locator Functionality" property of the "Connexis Capsule" stereotype to True.

When this stereotype property is set to True the "rTDBase" capsule part is renamed to "rTDBase_Locator" and retyped to RTDBase_Locator from the Connexis library. This capsule implements the Locator service. Note that it inherits from RTDBase so we still keep all the functionality provided by that capsule.

And finally, we need to specify which transport to be used for sending the messages between the applications. Connexis provides two kinds of transports out of the box, CRM and CDM. Let's use CDM for our application.

19. Select the Ping capsule and go to the Stereotypes tab in the Properties view. Set the "CDM Transport" property of the "Connexis Capsule" stereotype to True.
20. Repeat the same step for the Pong capsule.

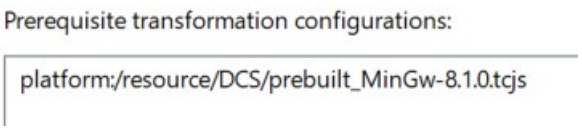
Build and Run the Model

The TC we used previously for building the application can not be used for building the updated model since it builds everything into a single application. We must create two new TCs, one for building an executable with Ping and one with Pong.

Right-click on the "Transformation Configurations" virtual folder and perform the command **Create Transformation Configuration File**. Do this twice, and call the first TC file "Ping.tcjs" and the second one "Pong.tcjs". Then set the following TC properties using the TC editor:

- On the Main tab, set the "Sources" property. Ping.tcjs should have it set to the packages Ping-Pong::Common and PingPong::PingPongClient while Pong.tcjs should have it set to the packages PingPong::Common and PingPong::PingPongServer.

- Also on the Main tab, mark the "Automatically create and update target project" checkbox and specify the "Workspace output path" for the target projects. Call it "/Ping_target" for Ping.tcjs and "/Pong_target" for Pong.tcjs.
- On the References tab, add a prerequisite TC for linking with the prebuilt Connexis library that is available in the Connexis library model. Choose one that matches the target platform you build for. For example:



- On the Code Generation tab, set the top capsule. Set it to the Ping capsule for Ping.tcjs and the Pong capsule for Pong.tcjs.
- On the Target Configuration tab, set the same target specific properties as you used for the "Default Transformation Configuration" previously. Also set the "Executable name" property to Ping$(EXEC_EXT) for Ping.tcjs and Pong$(EXEC_EXT) for Pong.tcjs.

Save the TCs and build them by using the context menu command **Build**.

Open two command prompts and run the built executables. Start Ping.exe like this:

```
Ping.exe -CNXep=cdm://localhost:7777 -CNXlpep=cdm://localhost:8888
-URTS_DEBUG=quit
```

Start Pong.exe like this:

```
Pong.exe -CNXep=cdm://localhost:8888 -CNXlp -URTS_DEBUG=quit
```

The `-CNXep` command-line option specifies the "endpoint location" for the executable (i.e. the machine where it is running and the port it listens to). The `-CNXlp` option specifies that the PongApp will act as the primary locator. The `-CNXlpep` option informs the PingApp of the location of the primary locator. For more information about Connexis command-line options, refer to [Connexis Command Line Options](#).

A short while after both applications have been started you should see output that indicates a successful sending of ping and pong between the two executables using Connexis:

```
dcs: CDM Transport : enabled
dcs: CDM listening at [cdm://10.0.75.1:7777]
dcs: connecting to primary locator at [cdm://localhost:8888]
dcs: backup locator not configured
dcs: metric service available

received a pong
received a pong
received a pong
received a pong
received a pong
```

```
dcs: CDM Transport : enabled
dcs: CDM listening at [cdm://10.0.75.1:8888]
dcs: locator running as primary
dcs: connecting to primary locator locally
dcs: backup locator not configured
dcs: metric service available

received a ping
received a ping
received a ping
received a ping
received a ping
```

Congratulations, you have now built your first distributed application using Connexis.

If you want, you can now experiment with running the executables on different machines. The only change that is needed for this is to use different command-line options where "localhost" is replaced with the machine name or IP address.

Summary and Comments

The PingPong model you just have built is of course very simple, but it still illustrates the main Connexis concepts and workflow. We started by creating a regular UML-RT model, with wired ports and connectors, and then converted it to a Connexis model using unwired ports that are registered programmatically. In a more realistic scenario it is often known beforehand that the application needs to be distributed, and in that case you would of course build it using Connexis already from the beginning.

Note that only the ports that connect the distributed parts of the application need to be unwired. Within each application it is recommended to use wired ports and connectors since they allow the communication paths to be visually shown in structure diagrams.

In general it's a good idea to think about distribution aspects already when designing the application, mainly because of performance. Many of the performance issues that you may encounter in a distributed application are a direct result of not partitioning your model properly. Remember that intra-thread messages are faster than inter-thread messages, which are faster than inter-process messages, which are faster than inter-machine messages.

In the tutorial we used the Locator service to resolve the registration strings for the unwired ports. If the nature of your application is such that you know the names of the endpoints that you want to communicate with (either through the use of an algorithmic mapping or by reading a configuration file), then explicit endpoint names can be used in the registration strings of the unwired ports in the model. This avoids the need to use the Locator service and therefore improves performance.

Finally, the beauty of Connexis is that once your unwired ports have been registerred, and Connexis has established the connections between them, then your application can send and receive messages on these ports in exactly the same way as with normal ports.

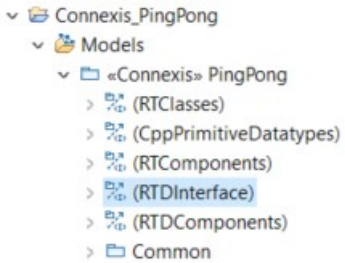
Connexis Services

The Connexis model library consists of a set of capsules, protocols, classes and data types. Together they implement a number of services which your UML-RT application can utilize in order to implement communication across process boundaries. That is, to realize a distributed application.

If you followed the Tutorial: Building a Distributed Application with Connexis you have already some experience of using a few of the services that are provided by Connexis. In this chapter we will go through all services provided by the Connexis library, and describe how you can use them in your model, and, where possible, how you can customize their behavior.

From a practical point of view, the use of a Connexis service typically means to add a few elements to your model (e.g. a capsule part or a port). Often several elements need to be added and configured properly. To make this easier you can take advantage of the Connexis profile. It abstracts the implementation details of the Connexis library into high-level settings, realized as stereotype properties. Rather than manually adding the necessary elements from the Connexis library to your model, you can apply stereotypes from the Connexis profile and set values for the properties of those stereotypes. The profile will then automatically update your model as necessary. The profile can both add necessary elements (when a new service should be used) or remove them (when an existing service no longer is needed). This speeds up the process of building a model with Connexis. However, if you prefer you can of course manually do the same things as the profile does.

The Connexis library elements that constitute the interface you should use in your application, are located in the RTDInterface package. If you use the Connexis profile this package is automatically imported in your model.



Otherwise you can find this package in the DCS model library at this location: DCS::Logical View::DCSModelInterfaces::RTDInterface.

Base Service

The Base service is a core service that each component that will use Connexis in your application must use. It is implemented by means of the RTDBase capsule of the Connexis library. This service is also the entry point for accessing most other Connexis services.

To use this service first select an appropriate capsule in each component of your application. Often it's a natural choice to select the top capsule of each component. Then apply the "Connexis Feature" stereotype on the selected capsules. Each selected capsule will then get a capsule part typed by the RTDBase capsule.

The "Connexis Feature" stereotype provides the following stereotype properties:

CDM Transport	Specifies whether CDM transport should be used in this Connexis component. See Transport Registration for more information.
CRM Transport	Specifies whether CRM transport should be used in this Connexis component. See Transport Registration for more information.
Locator Functionality	Specifies whether this Connexis component should contain a Locator. See Locator Service.

RTD InitStatus Port	Specifies whether this Connexis component will be notified about the initialization status of the Connexis library. See Initialization Status Service.
RTD Metrics Port	Specifies whether this Connexis component should use the Metrics Service.
Target Agent Functionality	Specifies whether this Connexis component should be instrumented to allow debugging of the distributed application. Note that debugging is done by means of the Connexis Viewer, which currently is not available in Model RealTime. It is available in Rational Rose RealTime. Also note that if you set this property to True, "CDM Transport" will also be set to True automatically since the Connexis Viewer uses the CDM transport.

Depending on how you choose the values for these stereotype properties, the type of the base capsule part will be set differently. It may either be RTDBase, or a capsule that inherits from RTDBase. The different configurations are summarized in the table below:

Configuration	Capsule	Description
Minimal	RTDBase	This is the minimal configuration where the component will not contain the Locator, nor the Target Agent for debugging with the Connexis Viewer. This configuration gives the smallest overhead in terms of the size of the executable.
Target Agent	RTDBase_Agent	This configuration includes the Target Agent but the Locator is not linked in with the executable.
Locator	RTDBase_Locator	This configuration includes the Locator but the Target Agent is not included.
Target Agent and Locator	RTDBase_Locator_Agent	This configuration includes both the Locator and the Target Agent. The configuration leads to the biggest overhead in terms of the size of the executable since both the Locator and the Target Agent is linked into the executable.

Transport Registration

A transport must register with Connexis prior to the initialization of the library. The default way of performing this registration is to do it when the Connexis Feature capsule is incarnated (which happens when the component starts up in case it is the top capsule). If you have configured your Connexis Feature capsule to use either CDM or CRM transport, the capsule will have an attribute typed by either RTDCdm or RTDCrm. The constructor of these classes perform the registration of the transports.

The default constructor will register the transport so that it listens for messages on the transporter's thread. This gives the best performance. However, only one transport can do this, so if your component uses multiple transports, the first one that is registered with Connexis will get this behavior. In order to be more explicit about which transport that should get this behavior, and not rely on the order in which the transport attributes are initialized, you can use instead the constructor that takes a boolean parameter. If you pass "false" to the constructor, the transport will instead use a separate thread for listening for messages.

If your component uses a custom transport, the initialization may be done in a different way, for example using a constructor with different parameters. However, any transport must be registered with Con-

nexis before the library is initialized, so it is in general a good idea to do it when the Connexis Feature capsule is incarnated.

Initialization Status Service

Before an application can use any Connexis feature, the Connexis library must have been initialized. The Initialization Status Service is the recommended approach for how the application can get notified when this initialization is completed.

When you set the stereotype property "RTD InitStatus Port" to "True" the Connexis Feature capsule will get a port "RTDInitStatus" typed by the RTDInitStatus protocol. This is an SAP port that is automatically registered with the registration string ":RTDInitStatus". The Connexis library contains a corresponding SPP port, and when the Connexis library has been fully initialized, these ports will be connected. This means that when the rtBound event arrives on the "RTDInitStatus" port, the Connexis initialization is completed, and the application can then start to use Connexis features.

If there are more capsules in the application that need to be notified when Connexis has been initialized, they can follow the same approach. Create a port in the capsule typed by the RTDInitStatus protocol and make sure the Notification property is turned on. If you prefer to register the port manually, set the Registration Kind property to "Application" and then register the port like this:

```
rTDInitStatus.registerSAP(":RTDInitStatus");
```

By default at most 50 clients can subscribe and get notified through this port. This limit is controlled by the constant DCS::Logical View::DCSComponents::DCSSysConfig::RTDConstants::rtdMaxStatus.

The RTDInitStatus protocol provides a number of Out Events and In Events that can be sent or received as soon as Connexis has been initialized. The table below describes these events:

Event	Direction	Description
rtdAgentActive	Out	Request for Target Agent activation status.
rtdAgentActiveReply (data : int)	In	Reply for the above event. The data tells whether the Target Agent is active (1) or not (0).
rtdBackupEndpoint (data : RTString)	Out	Set the endpoint of the backup locator. The data is the endpoint string, in the same format as the CNXlbep command line option. Examples: cdm://localhost:4000 tcp://192.139.251.2:5000
rtdBackupEndpointReply (data : int)	In	Reply for the above event. The data tells whether the request to set the backup locator endpoint was successful or not: 0 - success 1 - failed because this process is the backup locator 2 - failed due to an invalid endpoint string
rtdCDMport	Out	Request for the value of the CDM port assigned.
rtdCDMportReply (data : int)	In	Reply for the above event. The data is the CDM port, which was either specified by means of the CNXep command line argument, or automatically assigned to a free port number.

		A zero value indicates a software failure.
rtdDCSrunning	Out	Request for the initialization status of Connexis. There is usually no need to send this event, since when rtBound has been received, it's already known that Connexis has been initialized and is running.
rtdDCSrunningReply (data : int)	In	Reply to the above event. The data tells whether Connexis is initialized and is running (1) or not (0).
rtdLocatorAvailable	Out	Request for Locator availability status. If the Locator is present in the component and properly configured it will be considered as available.

rtdLocatorAvailableReply
(data : int)
	In	Reply to the above event. The data tells whether a Locator is available or not.
		A zero value implies that the locator is NOT properly configured and the registration of global names will fail:
		`port-name.registerSAP("dcs:/service-name") // will fail`
		`port-name.registerSPP("dcs:/service-name") // will pass since SPPs can also be connected locally and explicitly.`
		A non-zero value indicates that the Locator is available though a connection may not exist at this time to a remote Locator. Registration of global names will pass.
		In this case the value gives additional information about the status of the Locator:
		1 - Primary Locator running locally (this process), Backup Locator not configured
		2 - Primary Locator running locally (this process), Backup Locator is remote (CNXlbep)
		3 - Backup Locator running locally (this process), Primary Locator is remote (CNXlpep)
		4 - Primary Locator is remote (CNXlpep), Backup locator not configured
		5 - Primary Locator is remote (CNXlpep), Backup locator is remote (CNXlbep)

rtdPrimaryEndpoint	Out	Set the endpoint of the primary locator. The data is the endpoint string, in the same format as the CNXlpep command line option.
rtdPrimaryEndpointReply	In	Reply for the above event. The data tells whether the request to set the primary locator endpoint was successful or not: 0 - success 1 - failed because this process is the primary locator 2 - failed due to an invalid endpoint string
rtdTransportController	Out	Request for the Transport thread (i.e. a pointer to its Controller). High-performance Connexis applications where capsules are

		collocated on the same thread as the Transport can perform this request and incarnate the capsules on the returned Controller.
rtdTransportControllerReply (data : long)	In	Reply to the above event. The data is the address of the Controller for the Transport. Cast it to an RTController pointer: `RTController * t_thread = (RTController *)*rtdata;` A null pointer is returned if Connexis has not been initialized.
rtdVClimit	Out	Request for the limit on the maximum number of Virtual Circuits (VCs). This limit is defined by the constant DCS::Logical View::DCSComponents::DCSSysConfig::RTDConstants::rtd-MaxStatus.
rtdVClimitReply (data : int)	In	Reply to the above event. The data is the maximum number of Virtual Circuits that Connexis is configured to handle.

For the special case when the Connexis Feature capsule runs on the main thread in a fixed capsule part, it's not strictly necessary to use the Initialization Status Service, because in that case you can know that Connexis has already been initialized when the capsule state machine starts to run. However, it's still recommended to use the Initialization Status Service so that your application will continue to work even if you later decide to run the Connexis Feature capsule on another thread.

Locator Service

The Locator Service allows a distributed application to communicate without using explicit endpoint addresses (such as IP addresses and ports). The Locator is a name server and introduces an indirection where the application instead uses logical service names to refer to the endpoints that are connected.

The Locator Service supports both a primary and a backup locator. In this way, a distributed application can be made more robust by ensuring that the name server is not a single point of failure. The backup locator automatically takes over if the primary locator for some reason become unavailable (for example because its container executable has crashed or is non-responsive).

Depending on the nature of your application you may or may not benefit from the Locator Service. Here are some examples when it makes sense to use it:

- The application has to run on different machines
- The application needs a network topology that can be dynamically changed without affecting the currently executing software.
- The application needs to implement some kind of load sharing topology.

To use the Locator Service simply set the stereotype property "Locator Functionality" to "True". The RTDBase capsule part of the Connexis Feature capsule will then be retyped to RTDBase_Locator. If you also have set the "Target Agent Functionality" to "True" the type will instead be RTDBase_Locator_Agent. You have to perform this step in each component that should contain a Locator (either a primary or backup locator).

The use of the Locator Service has a very small impact on the application. The only difference is in the registration strings that are used when registering ports. They are on the following form:

```
dcs:/<service-name>
```

The service name can be any string that describes the service which the registered SPP port provides to connected SAP ports.

Publisher Ranking

Multiple SPP ports can use the same service name. In this case the Locator Service performs a ranking of all those ports. The port with the highest rank will be used for the binding when an SAP port is registered with the same service name. The rank of a publisher is specified in the registration string by means of a registration parameter. Here is an example:

```
dcs:/theService((locator_rank, 1))
```

The default rank of a publisher is zero.

Ranking makes it possible to dynamically replace a certain publisher with another one. Subscribers that are already bound to a publisher remain bound to it, but new subscribers will be bound to the other publisher since it has a higher rank.

If there are multiple registered publishers with the same rank the Locator Service will prioritize one that has a certain transport protocol, if a preferred transport has been specified. Such a preferred transport can be done using a registration parameter in the registration string. For example:

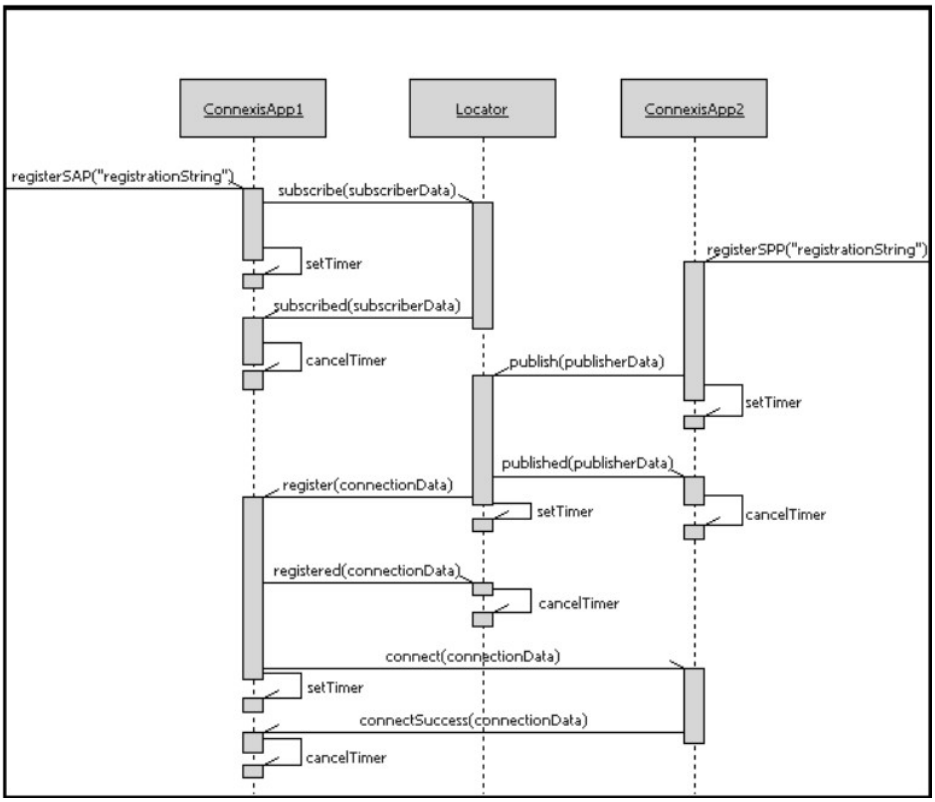
```
dcs:/theService((locator_transport, crm))
```

If there is no such preferred transport for the individual subscriber, the Locator Service will next check if the command-line option CNXlocator_preferred_transport has been used to specified a preferred transport.

As a last resort, if there still are multiple matching publishers a round-robin scheme is used to decide to which of those publishers the subscriber will be bound. In this case the binding of a subscriber to a publisher hence becomes non-deterministic.

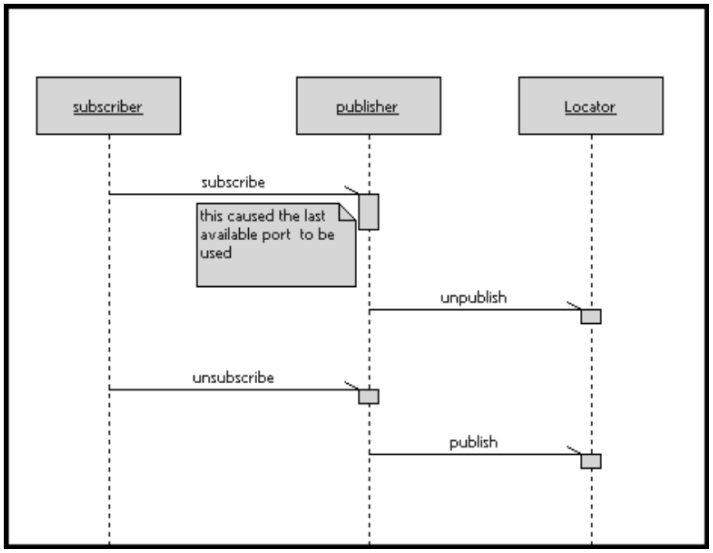
Locator Dynamics

From the perspective of the user application, the Locator Service is simple to use without knowing anything of how it works internally. However, for an increased understanding of the locator dynamics, let's look at what really happens when an SAP port gets bound to an SPP port by means of the Locator Service:



As can be seen in the diagram above, several internal messages are exchanged between the Connexis applications and the Locator to realize the binding. You should be aware of this overhead when designing your application. However, once the SAP port is bound to the SPP port there is no overhead introduced by the Locator Service. In fact, the Locator Service is not at all involved in the actual communication between the two applications which takes place after the ports have become connected.

Another thing to be aware of is what happens when a publisher becomes fully subscribed, i.e. when all the ports of the publisher (as determined by the port multiplicity) have been subscribed to. When this happens, there is not room for another subscriber to be bound to the publisher. The publisher therefore "unpublishes" itself from the Locator. Thereby a future subscriber can be bound to another publisher that is registered with the same service name, possibly with a lower rank. If one of the subscribers are deregistered from a fully subscribed publisher, the publisher once again publishes itself with the Locator. The diagram below illustrates what happens:



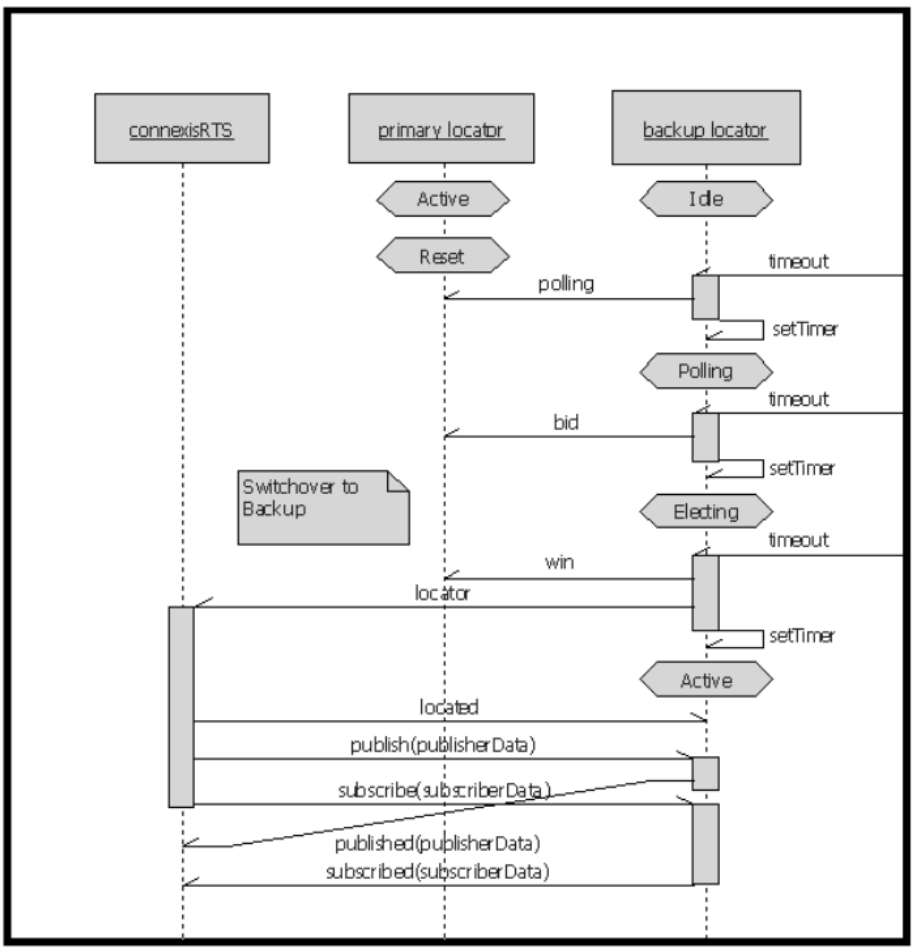
Backup Locator

If the distributed application uses a backup locator it should be placed in a different executable than the primary locator. It can then take over the job of the primary locator if it becomes unavailable (for example because its application crashes). You hence avoid a single point of failure in the application.

The backup locator continously polls the primary locator, and if it doesn't get a timely response it will assume the primary locator is unavailable and start the fail-over procedure. The backup locator broadcasts a message to all endpoints to inform that it now is the new primary locator. Each endpoint acknowledges this message by republishing all Connexis publisher ports, as well as all Connexis subscriber ports that are pending to be connected. This procedure takes some time, and in the meanwhile new registrations of SPPs or SAPs with locator registration strings will be delayed (but not lost) until the backup locator is ready to take over.

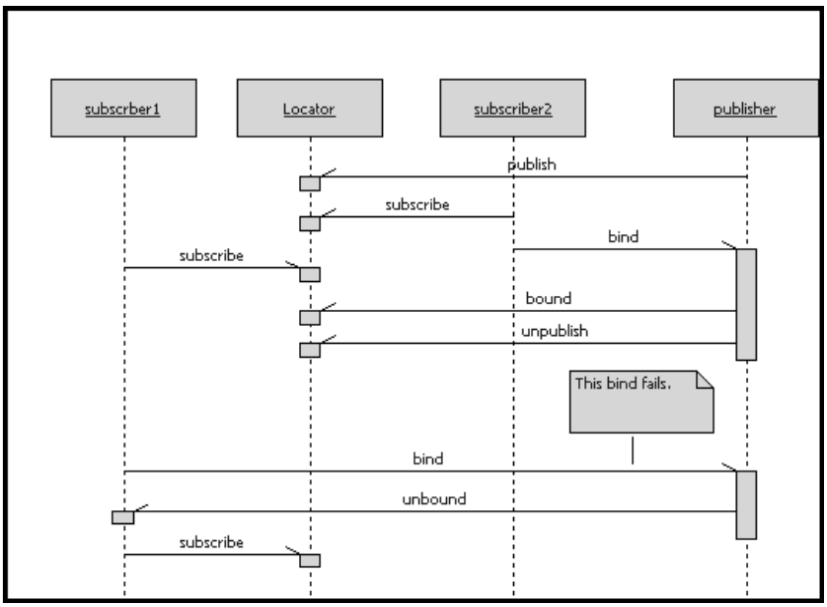
If the primary locator later becomes available again it will automatically take the role as the new backup locator. It is therefore usually enough with one level of locator backup.

The picture below illustrates the fail-over procedure when the backup locator takes over the responsibility of the primary locator:



Locator Race Condition

In the time that passes between the locator resolves the endpoint of a publisher and when a subscriber gets bound to that publisher it can happen that the publisher becomes unavailable, for example because it is deregistered. Such a scenario is illustrated below:



Here the locator does not get notified about the binding of subscriber2 until after subscriber1 is registered. Hence subscriber1 will attempt to bind to the publisher, but this will fail assuming that the publisher is an SPP port with multiplicity 1. In this case subscriber1 will simply subscribe again with the locator. The subscription will become pending until either a matching publisher becomes available, or the SAP port is deregistered so that it gets unsubscribed from the locator.

Locator Configuration

Command-line options are used for specifying the locator configuration. This includes information about which locator that should be the primary, and which that should be the backup. There are also a few other command-line options that can be used for specifying how the locator(s) should operate.

Read more about the available command-line options in Connexis Command Line Options.

The table below lists common locator configurations and the corresponding command-line options used:

Configuration	Command-line Options	Comment
When starting an executable that hosts the primary locator and no backup is being used.	CNXlp	The CNXlp option ("lp" = locator primary) establishes the process as the primary locator.
When starting an executable that hosts the primary locator and a backup is being used.	CNXlp, CNXlbep	The CNXlbep option ("lbep" = locator backup endpoint) informs about the existance and location of the backup locator.
When starting an executable that hosts the backup locator.	CNXlb, CNXlpep	The CNXlb option ("lb" = locator backup) establishes the process as the backup locator. The CNXlpep option ("lpep" = locator primary endpoint) informs about the existance and location of the primary locator.
When starting an executable that is using a primary locator with no backup.	CNXlpep	
When starting an executable that is using a primary locator with a backup.	CNXlpep, CNXlbep	

Below are some examples of starting executables that are part of a distributed application that uses the Locator Service:

Example 1: Two node application with no backup locator

To start the executable that acts as the primary locator:

```
<app1_name> -CNXep=cdm://host1:9999 -CNXlp
```

To start the other executable:

```
<app2_name> -CNXep=cdm://host2:9991 -CNXlpep=cdm://host1:9999
```

Example 2: Three node application with primary and backup locator

To start the executable that acts as the primary locator:

```
<app1_name> -CNXep=cdm://host1:9999 -CNXlp -CNXlbep=cdm://host2:9999
```

To start the executable that acts as the backup locator:

```
<app2_name> -CNXep=cdm://host2:9999 -CNXlb -CNXlpep=cdm://host1:9999
```

To start the third executable:

```
<app3_name> -CNXep=cdm://host3:9991 -CNXlpep=cdm://host1:9999
-CNXlbep=cdm://host2:9999
```

Customizing the Locator Service

The Locator Service is implemented in a general fashion to make it usable in many kinds of distributed applications. As already described the Locator Service has several features that make it more than a simple name service:

- it supports multiple publishers registered with the same service name, and implement an algorithm for choosing the most appropriate publisher for a subscriber based on publisher rank and preferred transport protocol
- it allows pending subscriptions that are automatically connected to matching publishers when they become available
- it provides automatic resubscription when a publisher becomes unavailable
- it implements a fail-over procedure to let a backup locator take over when the primary locator becomes unavailable

If you only want to do minor modifications to the behavior of the Locator Service you can simply copy the DCS model library and modify it according to your needs (see Customizing and Porting the Connexis Library). However, sometimes not all of the above features may be needed by a certain distributed application, and in that case you may choose to implement another mechanism for resolving service names to physical endpoint addresses.

One example could be to use service names that follow a certain pattern, so that the endpoint address can be derived directly from the service name. Another example could be to use a configuration file that all involved executables read at start-up. Thereby it becomes possible to do the name lookup locally in each executable which can give good performance.

Metrics Service

The Metrics Service can provide various metrics collected in a distributed application. Such metrics can help you identify performance bottlenecks and in general monitor the health of the application.

To use the Metrics Service you need to add a metrics port to a capsule and subscribe it to the Metrics Service. If you use the Connexis profile you can just set the stereotype attribute "RTD Metrics Port" to "True" in your "Connexis Feature" capsule. This will create the metrics port which is called "RTDMetrics" and is typed by the protocol RTDMetrics.

To subscribe the port to the Metrics Service register it with the following registration string:

```
dcs:RTDMetrics
```

This will connect the port to the Metrics Service of the local executable. However, you can also connect it to a Metrics Service of a remote executable in order to collect metrics from that executable. In that case use this registration string:

```
dcs://<host>:<port>/RTDMetrics
```

By default at most 50 clients can subscribe and get notified from each instance of the Metrics Service. This limit is controlled by the constant DCS::Logical View::DCSComponents::DCSSysConfig::RTD-Constants::rtdMaxStatus.

You can register the metrics port as soon as the application has started up, but you must wait for it to receive the rtBound event before you start to request any metrics.

The RTDMetrics protocol provides a number of Out Events and In Events that can be sent or received to collect metrics, turn on or turn off metrics collection, or clear collected metrics data. The table below describes these events:

Event	Direction	Description
rtdMetricsCollectOn	Out	Turn on metrics collection in the application that hosts the Metrics Service.
rtdMetricsCollectOnConfirm rtdMetricsCollectOnFail	In	Reply for the above event to indicate whether metrics collection was successfully turned on or not.
rtdMetricsCollectOff	Out	Turn off metrics collection in the application that hosts the Metrics Service.
rtdMetricsCollectOffConfirm rtdMetricsCollectOffFail	In	Reply for the above event to indicate whether metrics collection was successfully turned off or not.
rtdMetricsClear	Out	Clear all data collected by the Metrics Service.
rtdMetricsClearConfirm rtdMetricsClearFail	In	Reply for the above event to indicate whether collected metrics data was successfully cleared or not.
rtdMetricsInterval (data : int)	Out	Set the time interval for metrics collection. The data parameter specifies the number of seconds to wait between each metrics collection.
rtdMetricsIntervalConfirm (data : int) rtdMetricsIntervalFail (data: int)	In	Reply to the above event to indicate whether the metrics interval was successfully set or not. If the desired interval could not be set because it was too small, the minimal acceptable interval is set as the data. Otherwise the event data is the same as the interval that was requested.
rtdMetricsGet	Out	Request for metrics that have been collected up to this point in time.
rtdMetricsGetConfirm (data : RTDStats) rtdMetricsGetFail	In	Reply to the above event. In case of a successful request of metrics the rtdMetricsGetConfirm event will carry a data parameter typed by RTDStats. Refer to this class in the Connexis library for information about what metric data that is returned (DCS::Logical View::DCSModelInterfaces::RTDInterface::RTDStats).

In the prebuilt Connexis libraries that are shipped with Model RealTime the Metrics Service is by default enabled, but initially not turned on. If you want to turn on the collection of metrics before the metrics port has become bound you can use the special command-line option -CNXm=1. See Connexis Command-Line Options for more information.

If you never plan to use the Metrics Service, and want to optimize the size of the built executables, you can rebuild the Connexis library with the compile flag RTD_STATISTICS set to 0. See Customizing and Porting the Connexis Library.

Error Handling

Since Connexis binds registered ports asynchronously, error handling also is asynchronous. To get notified about errors related to a particular Connexis port, define a special In Event called "rtdError" in the protocol that types the port. This event should have a data parameter of type RTDErrorType. Each error that can occur is represented by one of the literals in the RTDErrorType enumeration:

Output	Description
rtdDCSUninitialized = 1	Registration failed because Connexis was not initialized.
rtdZeroReplication = 2	Registration failed because the multiplicity of the port is zero.
rtdInvalidSyntax = 3	Registration failed because the registration string had an invalid syntax. See Registration String Grammar for the expected syntax.
rtdInvalidTransport = 4	Registration failed because the specified transport is not supported by this component.
rtdCircuitUnavailable = 5	Registration failed because no virtual circuit is currently available for the remote binding.
rtdLocatorUnavailable = 6	Global registration failed because no locator is available.
rtdConnectTimeout = 7	Explicit registration failed because a connection could not be established with the remote endpoint in a timely manner.
rtdEndpointUnavailable = 8	A connection cannot be made at present with the remote endpoint because it is currently unavailable.
rtdEndpointInaccessible = 9	A connection can never be made with the remote endpoint.

If an error occurs the "rtdError" event will be sent to the port at General priority. The exception is the last two errors which can occur even if the registration and binding was successful, and they are therefore sent at Background priority.

One common error in a distributed application is that a subscriber that has successfully connected to a publisher later will loose that connection. The reason can for example be that the publisher is deregistered, or that the application where the publisher resides terminates. Depending on the transport that is used such a broken connection may or may not lead to an error message. If the transport that is used does not provide this "quality of service", you have to use another mechanism to detect such problems. A common solution is to let the subscriber send a certain "are-you-alive?" event periodically to the publisher. The publisher should respond to this event within a certain time limit. If the subscriber does not get the response within that time limit, it will assume that the publisher is no longer available and can then deregister itself. It may then choose to wait for some time and try again, or it can register itself with another service name, representing a backup service to be used.

Registration String Grammar

The syntax of Connexis registration strings are specified by the below grammar.

```
<dcs registration string> ::= dcs:[[<endpoint>]/]<service name>[(<option list>)]
```

```
<option list> ::= <option> [ <option list> ]

<option> ::= (locator_rank, integer) | (locator_transport, transport) |
(connect_retries, integer)

<endpoint> ::= <cdm endpoint> | <custom endpoint>

<cdm endpoint> ::= cdm://<host>:<port>

<crm endpoint> ::= crm://<host>:<port>

<host> ::= host name | ip address

<custom endpoint> ::= <protocol name>://<address>

<service name> ::= <name>

<protocol name> ::= <name>

<address> ::= <restricted string>

<port> ::= integer between 0 and 65535

<name> ::= alphanumeric string with optional underscores ("_")

<restricted string> ::= string without

       - comma ","

       - parenthesis "(" or ")"

       - white space
```

Connexis Command-Line Options

Connexis uses command-line options for specifying the configuration of the distributed application and for various settings used by the library. If you need to implement additional preferences for your application you can introduce additional custom command-line options at the end of the command-line, or use some other mechanism for information passing such as environment variables or a configuration file.

Each command-line option has a name and an alias (i.e. short name). Below the most commonly used command-line options are described. For a full list of all available command-line options refer to the Connexis library class DCS::Logical View::DCSComponents::DCSSysConfig::RTDConfigDefaultData.

General Options

General command-line options.

Command Line Option	Description
CNXunique_id (CNXui)	An identifier for the executable which should be unique within the distributed application. Choose a name that describes the role played by the executable in the application. *Argument Type*: string *Default Value*: a randomly generated id
CNXnobanner (CNXnb)	Suppresses the printing of a banner at start-up. This banner may contain diagnostic messages or initialization errors.

CNXdump (CNXd)	Prints the actual values of all command-line options that will be used. This includes what has been specified on the command-line as well as default values.
CNXhelp (CNXh)	Prints information about all available command-line options.

Transport Options

Command-line options related to transports.

Command Line Option	Description
CNXendpoint (CNXep)	Sets the endpoint for a transport. The endpoint is a string on the following format: `<transport>://<address>` If the transport is omitted it defaults to CDM. If the address just specifies a port and no hostname, the hostname defaults to localhost. *Argument Type*: string *Default Value*: none
CNXcdm_max_rx_size (CNXcmrs)	Configures the maximum CDM receive message size. *Argument Type*: int *Default Value*: Derived from the size of the biggest buffer in the buffer pool (see CNXtran_buffer_pool).
CNXcdm_udp_rx_size (CNXcurs)	Configures the maximum UDP receive message size. *Argument Type*: int *Default Value*: Depends on the operating system used.
CNXcdm_udp_tx_size (CNXcuts)	Configures the maximum UDP transmit size. *Argument Type*: int *Default Value*: Depends on the operating system used.
CNXtran_buffer_pool (CNXtbp)	Configures the buffer pool. For each buffer in the pool this option specifies the buffer size followed by the number of buffers. *Argument Type*: string *Default Value*: 64:1,600:10,4200:10,17000:2,32860:2,33000:2
CNXtran_first_msg_size (CNXtfms)	Configures the first message (i.e. initial buffer) size when encoding. *Argument Type*: int *Default Value*: 600
CNXtran_max_tx_size (CNXtmts)	Configures the maximum transmit message size for a transport. *Argument Type*: int *Default Value*: none

Locator Options

Command-line options for configuring the Locator Service.

Command Line Option	Description

CNXlocator_primary (CNXlp)	Specifies that this process should be the primary locator. *Argument Type*: none *Default Value*: none
CNXlocator_backup (CNXlb)	Specifies that this process should be the backup locator. *Argument Type*: none *Default Value*: none
CNXlocator_primary_ endpoint (CNXlpep)	Specifies the endpoint of the primary locator. This option must be specified for all executables in the distributed application, except the one that hosts the primary locator. *Argument Type*: string *Default Value*: none
CNXlocator_backup_ endpoint (CNXlbep)	Specifies the endpoint of the backup locator. This option must be specified for all executables in the distributed application, except the one that hosts the backup locator. *Argument Type*: string *Default Value*: none
CNXlocator_retry_delay (CNXlrd)	Specifies the amount of time, in milliseconds, to wait before retries. The value must be >50. *Argument Type*: integer *Default Value*: 1000
CNXlocator_audit_delay (CNXlad)	Specifies the amount of time, in milliseconds, to wait between audits of the primary and backup locators. The value must be >50. Together with CNXlocator_audits_oos this option controls how long the backup locator will wait until it considers the primary locator unavailable and starts the fail-over procedure. *Argument Type*: integer *Default Value*: 2000
CNXlocator_audits_oos (CNXlao)	Specifies the number of failed audits required to take the primary locator out of service. For example, if the value of this option is 3, the primary locator will be taken out of service after the third consecutive audit has failed. *Argument Type*: integer *Default Value*: 3
CNXlocator_preferred_ transport (CNXlpt)	Specifies the preferred transport protocol to be used in case there are multiple publishers with the same rank. This option can only be set for the primary or backup locators. *Argument Type*: string *Default Value*: none

Metrics Options

Command-line options for configuring the Metrics Service.

Command Line Option	Description
CNXmetrics (CNXm)	Specifies whether metrics should be collected at application start up, before the Metrics Service SPP port has become bound. Set it to 1 to collect metrics at start-up. *Argument Type*: boolean int *Default Value*: 0 (false)

Connexis Messages, Errors and Warnings

Connexis prints various messages, errors and warnings at run-time to help debugging and troubleshooting problems. There are three categories of such output:

- Informational initialization messages (printed by Connexis when an application starts up)
- Initialization errors (printed by Connexis when some error occurs while an application starts up)
- Parameter errors (printed by Connexis if command-line options are incorrectly used)

Initialization Messages

The table below lists several common initialization messages:

Output	Description
dcs: transport listening at [endpoint] The transport could be cdm, crm or your own customized transport.	This is output once the transport starts up and begins listening at the endpoint. If it does not appear, your transport was probably not included.
dcs:CNXcmrs set to [size]	The CDM maximum receive size specified was larger than the largest buffer available. Check your use of the command-line options CNXtran_buffer_pool and CNXcdm_max_rx_size.
dcs:CNXtfms set to [size]	The first message size specified was larger than the largest buffer available or the max message size. Check your use of the command-line options CNXtran_max_tx_size, CNXtran_first_msg_size and CNXtran_buffer_pool.
dcs:***** CNXendpoint port not specified - free port will be selected *****	A port was not specified for the endpoint and Connexis will therefore choose a free unused port on which the CDM transport will listen. If you want the transport to use a specific port, use the command-line option CNXendpoint.
dcs: target agent enabled	Indicates that the target agent is running. The target agent must be running if you want to use the Connexis Viewer.
dcs: locator running as primary	Indicates that the locator was linked into the executable and is configured as the primary locator.
dcs: locator running as backup	Indicates that the locator was linked into the executable and configured as the backup locator.
dcs: local locator not running (CNXlp or CNXlb required)	You are using the RTDBase_Locator or the RTDBase_Locator_Agent capsule in your model. The locator was linked into the executable but has not been configured. Use the command-line option CNXlocator_primary or CNXlocator_backup to configure the locator.

dcs: locator service not available	The locator is not available based on configuration parameters.
dcs: metric service enabled	Indicates that the metrics service is enabled.
dcs: connecting to primary locator at [endpoint] dcs: connecting to backup locator at [endpoint]	These two lines are output as a pair and indicate that the primary locator is remote (see command-line option CNXlocator_primary_endpoint) and the backup locator is remote as well (see command-line option CNXlocator_backup_endpoint).
dcs: ***** Parameter [<old parameter name>] not supported. Please use [<parameter name>=<value>] *****	Indicates that an obsolete command-line option that is not supported with the current release has been used. When Connexis encounters use of such a command-line option it internally converts it to the new format and outputs this information message. You should switch to using the recommended command-line option instead.

Initialization Errors

In case of an initialization error, Connexis prints a general error message that looks like this (this banner will not be printed if the command-line option CNXnobanner has been set):

```
dcs: ********************************************************
dcs: ***** initialization failure - dcs not available *****
dcs: ********************************************************
dcs: ***** banner provided for diagnosis purposes ****
dcs: ***** use CNXd to display configuration *****
dcs: ********************************************************
dcs: !!!!! system failure when initializing the : <step> (<error>) !!!!!
```

<step> is one of the following:

- **configuration** problem in parsing parameters
- **target agent** refers to target agent for Viewer
- **transport-capsule**: refers to transport router
- **transport-callback**: refers to the transport callback thread (input)
- **transport-helpers**: refers to the transport helper thread (output)
- **controller**: refers to registration control
- **locator**: refers to the Connexis locator service
- **system**: any other general error

Most of these errors are internal errors that should never happen. If they still happen it could sometimes be because not enough system resources were available which caused Connexis to fail. For some messages there will be additional information printed, and in all cases there is an error code printed..

Parameter Errors

The table below lists error messages that are related to incorrect or inconsistent use of command-line options:

Output	Description
dcs: ***** multiply defined parameter [<name>=<value>] ignored *****	Reported if you try to use a command-line option multiple times.
dcs: ***** unknown parameter	A command-line option you have specified is not valid. This check is

[<name>=<value>] ignored *****	performed for all command-line options starting with CNX. See Connexis Command-Line Options for the list of valid command-line options, or use the CNXhelp command-line option.
dcs: ***** CNXendpoint invalid port [<value>] *****	The port number specified for the command-line option CNXendpoint is invalid (non-numeric or out of range).
***** CNXendpoint (CNXep) invalid port [port #] - freeport will be selected ****	The endpoint specification contains a syntax error. Connexis will choose a free port on which to listen.
dcs: ***** # of mblks less than # of buffers in buffer pool *****	The Connexis Transport buffer pool is not setup properly. Check your use of the command-line option CNXtran_buffer_pool.
dcs: ***** Not enough buffers in buffer pool specified *****	
dcs: ***** invalid buffer pool specified *****	
dcs: ***** CNXcurs = UDP system receive buffer size must be > max receive msg size - using target default *****	UDP buffers are not properly configured. Check your use of the command-line option CNXcdm_udp_rx_size. Connexis will use the default UDP receive buffer size which depends on the operating system that is used.
dcs: ***** CNXcuts - UDP system Tx buffer size smaller than max buffer size defined in buffer pool - using system default *****	UDP buffers are not properly configured. Check your use of the command-line option CNXcdm_udp_tx_size. Connexis will use the default UDP send buffer size which depends on the operating system that is used.
dcs: ***** CNXlpep ignored (CNXlp takes precedence over CNXlpep) *****	This error occurs if you specify that the executable should host the primary locator, and at the same time specify that the primary locator is in another executable. This is inconsistent, and the command-line option CNXlocator_primary_endpoint will be ignored.
dcs: ***** CNXlb ignored (CNXlp takes precedence over CNXlb) *****	An executable can either host a primary locator or a backup locator but not both at the same time.
dcs: ***** CNXlbep ignored (CNXlb takes precedence over CNXlbep) *****	This error occurs if you specify that the executable should host the backup locator, and at the same time specify that the backup locator is in another executable. This is inconsistent, and the command-line option CNXlocator_backup_endpoint will be ignored.
dcs: ***** CNXlp ignored (locator not present) *****	You are not using the RTDBase_Locator nor the RTDBase_Locator_Agent capsule in your model which means the locator is not linked into the executable. Still the command-line option CNXlocator_primary was used to configure the primary locator (which does not exist).
dcs: ***** CNXlb ignored (locator not present) *****	You are not using the RTDBase_Locator nor the RTDBase_Locator_Agent capsule in your model which means the locator is not linked into the executable. Still the command-line option CNXlocator_backup was used to configure the backup locator (which does not exist).
dcs: ***** CNXlpep missing (CNXlpep mandatory at backup locator) *****	When launching the executable that hosts the backup locator you must specify the location of the primary locator. Use the command-line option CNXlocator_primary_endpoint.
dcs: ***** CNXlpep missing	

(CNXlpep mandatory when using backup locator) *****	

Customizing and Porting the Connexis Library

The Connexis library is tightly integrated with the TargetRTS (a.k.a. the RT Services Library). The Model RealTime installation contains prebuilt versions of the Connexis library for the same platforms as for the TargetRTS (except MSVS x64). If you need to build and run your distributed application on another platform you need to build both the TargetRTS and the Connexis library for that platform.

Another reason for building the Connexis library, instead of using one of the prebuilt libraries, is if you want to customize its behavior or optimize it. For example:

- Changing the compiler flags used for building the library (for example to set some flags that will remove features you don't need)
- Implement your own name service to be used instead of the Locator service
- Implement a custom transport

Whatever the reason is for building the Connexis library, it's recommended to start with copying the Connexis model library (DCS) so you can modify the model and/or the TCs it contains.

Porting Connexis to a New Target Configuration

These are the steps to perform when you need to build Connexis for another target platform:

1. Build the TargetRTS for the new target configuration
2. Create Connexis specific header files for the new target configuration
3. Create a new library TC for the Connexis model
4. Configure CDR encoding/decoding for the new target configuration
5. Build and test the new library TC

Build the TargetRTS for a New Target Configuration

The Connexis library for a target platform depends on the TargetRTS for its target configuration and library settings. The TargetRTS provides several settings that can be configured. The table below lists those settings that are important for Connexis.

Target Setting	Value	Descriptions
USE_THREADS	1	Connexis is only available for multi-threaded applications.
HAVE_INET	1	Connexis requires IP support for the Connexis Datagram Messaging (CDM) and the Connexis Reliable Messaging (CRM) transports.
OBJECT_ENCODE	1	Messages must be encoded before they can be transmitted over the wire.
OBJECT_DECODE	1	A message received over the wire must be decoded into an object.

Compiler settings that are common to both the TargetRTS, the Connexis library and the application itself should be configured using the LIBSETCCFLAGS macro in $RTS_HOME/libset/<libset>/libset.mk.

Compiler settings that only apply to the TargetRTS should be set in the LIBSETCCEXTRA macro in $RTS_HOME/libset/<libset>/libset.mk.

Compiler settings that only apply to the Connexis library should be configured in the library TC used for building the Connexis model. See Create a C++ Library TC for Connexis.

For more information about how to build the TargetRTS for a new target configuration, refer to the chapter "Porting the TargetRTS" in the document "The RT Services Library - How to manage it using the TargetRTS wizard".

Creating Connexis Target Specific Header Files

Although most of the configuration of the Connexis library is done within the Connexis model, the Connexis thread configurations are configured in the file $RTS_HOME/target/<target>/RTDcsTarget.h. This file contains specific operating system priority definitions and configuration of the stack size for Connexis threads. The RTDcsTarget.h file also contains the definitions of the maximum and minimum values of the thread priorities for an operating system. Run-time argument processing uses these values to validate the run-time settings or the thread priorities for the Connexis threads. The table below provides a list of constants that must be defined in RTDcsTarget.h.

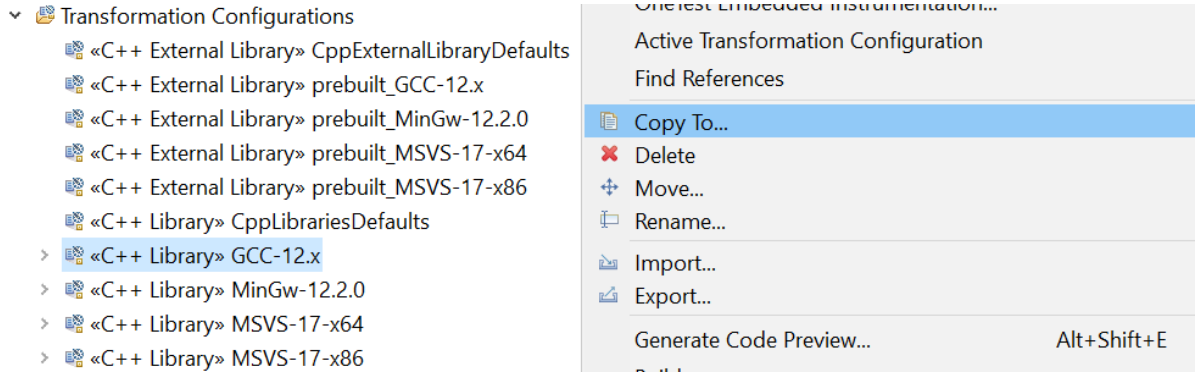
Constant	Description

CNX_PRIORITY_MIN
CNX_PRIORITY_MAX
	Defines the minimum and maximum allowable value for a thread priority. The CNX_PRIORITY_MIN and CNX_PRIORITY_MAX are used by Connexis to do bounds checking of thread priorities (for example, to determine whether the given priority is a legal value). They should be set to the lowest and highest numerical, not logical, thread values as defined by the operating system.
	For example:
	For operating system A, the highest thread priority is 10 and the lowest thread priority is 0. For operating system B, the highest thread priority is 0 and its lowest thread priority is 10. For both operating systems, CNX_PRIORITY_MIN and CNX_PRIORITY_MAX should be defined as:
	`#define CNX_PRIORITY_MIN 0`
	`#define CNX_PRIORITY_MAX 10`

CNX_PRIORITY_RAISE	By default the priority of the helper threads are higher than the priority of the transporter thread. The CNX_PRIORITY_RAISE constant can be used to increment a thread priority. For OS targets in which the higher priority threads have lower numeric values, the value of CNX_PRIORITY_RAISE must be negative.
DEFAULT_CNXTTP_PRIORITY	Sets the default thread priority for the transporter thread.
DEFAULT_CNXHTP_PRIORITY	Sets the default thread priority for the helper threads. For optimal system performance, the helper threads should run at a higher priority than the transporter thread.
DEFAULT_CNXATP_PRIORITY	Sets the default thread priority for the target agent. The target agent is designed to be minimally intrusive to the application and should be running at a lower priority than the threads of the application.
CNXTTP_STACK	Sets the stack size for the transporter thread.
CNXHTP_STACK	Sets the stack size for the helper threads
CNXATP_STACK	Sets the stack size for the target agent thread.

Create a C++ Library TC for Connexis

If you take a copy of the Connexis model library (DCS) you can either modify an existing library TC or create a completely new TC for building the library. You can clone an existing TC by means of the context menu command **Copy To**.



The table below lists the TC properties that typically need to be set in order to build Connexis for a new target configuration:

TC Property	Description
TargetRTS configuration (TargetConfiguration tab)	The <target> and <libset> settings of the TargetRTS configuration that you are building against. Set the property "Target services library" to the folder where you have already built the TargetRTS.
Make type (TargetConfiguration tab)	Specify the dialect of the makefiles to be generated. For example, if you are using the GNU make utility in your environment, this property should be set to GNU_make.
Make command (TargetConfiguration tab)	Specifies the name of the make utility to be used.
Make arguments (TargetConfiguration tab)	Here you must specify RT_SRC_TGT=<target_base>. Connexis depends on TargetRTS files in the $RTS_HOME/src/target directory. For example, RTtcp.h. The RT_SRC_TGT make variable is used to specify the target base.
Compile arguments (TargetConfiguration tab)	If you get compilation errors when building your TC it may be a result of the RTD_CONNEXIS_BUILD constant not being set properly. The definition of this constant should be changed from $(CNX_BUILD_NUM) to a user-defined integer value. This property is used to configure the C++ preprocessor macros that configure certain Connexis capabilities. Viewer tracing is configured on the target using the RTD_TRACE macro. $(DEFINE_TAG)RTD_TRACE=1 enables tracing. $(DEFINE_TAG)RTD_TRACE=0 disables most traces (except errors and warnings). $(DEFINE_TAG)RTD_TRACE=2 disables all traces. The metrics collection and reporting capabilities are configured using the RTD_STATISTICS macro. $(DEFINE_TAG)RTD_STATISTICS=1 enables metrics collection and reporting. $(DEFINE_TAG)RTD_STATISTICS=0 disables the metrics collection and reporting. Additonal macros might be required to configure the CDR encode/decode capabilities for the target platform. These capabilties are described in

	Configure CDR Encoding/Decoding for a New Target Configuration.

Connexis by default has plenty of tracing and debugging capabilities included. This is useful during the development of a distributed application, but when you are ready to productify the application you may want to exclude such capabilities to reduce the memory footprint of Connexis. These two compile arguments allow you to accomplish this:

- $(DEFINE_TAG)RTD_TRACE=0
- $(DEFINE_TAG)RTD_STATISTICS=0

Configure CDR Encoding/Decoding for a New Target Configuration

The CDR encode/decode functionality could require platform specific customizations depending on the capabilities of the platform. These customizations are accomplished by defining C++ preprocessor macros in the "Compile arguments" TC property. The following customizations are available:

- Overriding the type for use when encoding 64-bit values. The default behavior when the RTD_LONGLONG_TYPE macro is not defined is to encode/decode 64-bit values using the primitive "long long" type. This customization is required when the compiler does not provide support for "long long" types. Setting $(DEFINE_TAG)RTD_LONGLONG_TYPE=0 will use the "__int64" type to encode/decode 64-bit values. Setting $(DEFINE_TAG)RTD_LONGLONG_TYPE=1 will cause 64-bit values to be encoded/decoded using the primitive "double" type.
- Enabling the inclusion of <sys/types.h>. Some platforms require this inclusion to provide definitions of all the system types. Setting $(DEFINE_TAG)RTD_INCLUDE_TYPES_IN_RTDPLATFORMCONFIG enables this capability.

Build and Test the New Library TC

Once your library TC is ready, just build it as any other library. Set it as a prerequisite of all TCs that you use for building all executables of your distributed application that run on the specific target. You may have other executables that run on different targets that then would link with another build of the Connexis library.

To do a first basic test of your newly built Connexis library you can for example use the HelloWorld sample that is included with Model RealTime. See The Connexis HelloWorld sample model.