

SAP .NET Connector 3.1 Overview



HELP.BCMIDCONF

Release 3.1



Copyright

© Copyright 2022 SAP SE. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Excel, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, System i, System i5, System p, System p5, System x, System z, System z10, System z9, z10, z9, iSeries, pSeries, xSeries, zSeries, eServer, z/VM, z/OS, i5/OS, S/390, OS/390, OS/400, AS/400, S/390 Parallel Enterprise Server, PowerVM, Power Architecture, POWER6+, POWER6, POWER5+, POWER5, POWER, OpenPower, PowerPC, BatchPipes, BladeCenter, System Storage, GPFS, HACMP, RETAIN, DB2 Connect, RACF, Redbooks, OS/2, Parallel Sysplex, MVS/ESA, AIX, Intelligent Miner, WebSphere, Netfinity, Tivoli and Informix are trademarks or registered trademarks of IBM Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group. Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, R/3, SAP NetWeaver, Duet, PartnerEdge, ByDesign, SAP Business ByDesign, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE in Germany and other countries.






Business Objects and the Business Objects logo, BusinessObjects, Crystal Reports, Crystal Decisions, Web Intelligence, Xcelsius, and other Business Objects products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of Business Objects S.A. in the United States and in other countries. Business Objects is an SAP company.

All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP SE and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and

services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Icons in Body Text

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help → General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
<i>Example text</i>	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
Example text	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
Example text	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
EXAMPLE TEXT	Keys on the keyboard, for example, F2 or ENTER.

Content

Introduction	5
Installation.....	5
Providing Logon Parameters	5
Working With Data Containers and the Repository	6
RFC Client Programs	8
Synchronous RFC	9
Stateful RFC Client	11
tRFC/qRFC/bgRFC.....	12
RFC Server Programs.....	14
Synchronous RFC	16
Stateful RFC Server.....	18
Your .NET program does not have its own user session concept	18
Your .NET program has its own user session concept	19
tRFC/qRFC/bgRFC.....	20

Introduction

The following document will give you an overview of the architecture of the SAP .NET Connector 3.1 (NCo 3.1). It will describe the high-level concepts that you need to be familiar with in order to write .NET programs using NCo 3.1 successfully. It won't talk about the exact details and features of every NCo class – for that you will need to refer to the NCo API documentation – but the current document will give you a good understanding of the general features of NCo 3.1 and how to best use them in your own programs.

Please note: all classes mentioned in the following, use the namespace `SAP.Middleware.Connector`.

Installation

You can find detailed information on installation requirements for SAP .NET Connector 3.1 in SAP note [3152653](#).

Providing Logon Parameters

When designing the component that handles logon data, a special emphasis was placed on security concerns. The goal was to make the following three attacks as hard as possible:

- an unauthorized person reading sensitive logon data (e.g., logon data stored along with the application in a configuration file)
- a malicious application (which may be running inside a larger application context or framework) replacing the configured logon data with its own logon data to obtain backend user sessions with different authorizations from the ones that were originally intended for this application.
- a malicious application (which may be running inside a larger application context or framework) getting access to an open connection that was originally intended for different applications (which are running inside the same larger context).

Of course, it is still possible to provide logon parameters (for client access to the ABAP backend system as well as for starting RFC servers) in the `app.config` file of the application to allow fast and easy demo and proof-of-concept applications. But if you want to follow a different approach in productive setup (which is strongly recommended), for example reading user information on the fly from an LDAP directory, you can use the following mechanism (let's first concentrate on the RFC client case):

Let's assume no destination information is customized in the `app.config` file of the current executable. Then, whenever the .NET Connector needs destination data including logon information for opening an RFC connection into some ABAP system, it tries to make a callback into your code and ask for that information. In order to provide this callback, you need to implement the interface `IDestinationConfiguration` and hand an instance of that class over to the .NET Connector during startup of your application. The necessary steps for this look as follows:

- Implement the method `IDestinationConfiguration.GetParameters(string destinationName)`. Inside this method you can do anything you like, e.g. read the necessary logon parameters from a database or LDAP directory, open a GUI dialog box to prompt the current user for his credentials, or even hard-code the parameters in this method (although this is not exactly good style...). If logon parameters for a system called `destinationName` are not known, then the method just needs to return `null`.
- Create an instance of the above implementation and hand it over to NCo using `RfcDestinationManager.RegisterDestinationConfiguration()`.
- Start making RFC client calls (as will be described in chapter [RFC Client Programs](#)), and NCo will automatically obtain logon information for each backend system, as needed, by calling

the `GetParameters()` method of your `IDestinationConfiguration` implementation.

Note that once the .NET Connector has initialized a connection pool for a given RFC destination, it will no longer need the configuration data for that destination and consequently will not ask your implementation for them anymore. So what if an administrator wants to change some of these parameters later on, but doesn't want to restart the program? For example suppose he wants to turn on RFC trace on the fly in order to investigate a problem with a certain function module? Or he realizes that the traffic is higher than expected today, and therefore he wants to increase the connection pool size for that destination? For this purpose the `IDestinationConfiguration` interface contains one more feature - eventing:

```
public event RfcDestinationManager.ConfigurationChangeHandler ConfigurationChanged;
```

The `RfcDestinationManager` will register for this event at your instance of `IDestinationConfiguration`. So whenever you want to change some of the logon or connection parameters, your `IDestinationConfiguration` implementation just needs to trigger this event with the updated parameters, and the `RfcDestinationManager` will immediately change all connections in its pool accordingly. (Currently ongoing RFC calls, however, will not be interrupted.)

Without eventing, the configuration parameters will be checked only when fetching a destination again using `RfcDestinationManager.GetDestination()`.

If you need even finer control over logon parameters (for example if the application shall also cover the scenario of a large number of users logging on on-the-fly for only one or two calls, or if it is supposed to be using SSO or X509 certificate logon mechanisms, which require a frequent ticket update), you will probably need to work with the class `RfcCustomDestination`. But this topic is beyond the scope of this overview document.

Configuration parameters for RFC server programs are maintained in a completely analogous way to the one described above. The difference is that you have to implement the interface `IServerConfiguration` and work with the class `RfcServerManager` instead. Also, for server configurations it is possible to update a configuration for a concrete server using events in case the configuration data is changed.

Working With Data Containers and the Repository

Data containers are objects that hold a function module's IMPORTING, EXPORTING, CHANGING and TABLES parameters to be exchanged with the ABAP side. Working with them should be more or less self-explanatory. However, as this is one of the areas that are substantial to the programming model of NCo 3.1, they are describe in more detail.

In .NET Connector 3.1, function modules, structures and tables are "assembled" dynamically at runtime from information obtained from the backend's data dictionary (DDIC). Any function module, structure and tables is represented by an instance of the classes `IRfcFunction`, `IRfcStructure` or `IRfcTable`, which contain a map or tree hierarchy of all fields and subfields, some of which get created on the fly, when first accessed. These objects are called "Data Container", and they are created dynamically from blueprints obtained from the so called "Repository". So basically, instead of hard-wiring everything statically at design time, NCo 3.1 handles everything dynamically at runtime. Another advantage of that approach is that you don't need to create the objects for structures and tables yourself in your application code. You only create one `IRfcFunction` object instance, and it internally takes care of all contained structures and tables (using lazy initialization).

Here is some sample code of how this could look like:

```
RfcRepository rep = ...; // (How to get a repository, is covered
                        // later in chapter RFC Client Programs
IRfcFunction companyBapi = rep.CreateFunction(
                        "BAPI_COMPANY_GETDETAIL");
companyBapi.SetValue("COMPANYID", "001000");
// Execute the function in the backend. This is again described
// in chapter RFC Client Programs.
IRfcStructure detail =
                        companyBapi.GetStructure("COMPANY_DETAIL");
String companyName = detail.GetString("NAME1");
```

Now you may ask yourself, "how does NCo (or rather the RfcRepository object) know how the function module BAPI_COMPANY_GETDETAIL looks like?" For this, there are three different options:

1. You create a repository with logon information to the backend system. (These logon credentials need only limited authorizations and can be different from those user credentials used for executing the actual business functions in the backend, see also SAP note [460089](#).) Whenever the repository encounters an unknown function name, it executes a call into the backend's DDIC and looks up the signature of that function module, including all structure definitions for its parameters and tables. After that the information is cached, so no further backend lookups are necessary if the function is needed several times.

The advantage of this mechanism is: the .NET Connector side always works with up-to-date information. You no longer need to regenerate coding and recompile your application, if one of your function module parameters is enlarged from CHAR10 to CHAR20, or if the backend system is upgraded from non-Unicode to Unicode, etc.

The disadvantage is of course a slight overhead for the very first function call in the lifetime of an application.

2. You can create repositories with your own hard-coded meta-data information describing the necessary function modules and their structures/tables.

Basically, this is equivalent to an approach with typed proxies, so people who don't want to use the new dynamic approach as described in option 1, can fall back to this option. However, there are hardly any situations in which this would make sense. It is more tedious and more error-prone in case the backend side definition is changed. The only use cases that indeed make sense:

- an RFC server application providing function module names that do not exist on ABAP side
- an RFC server application that shall not need client-side configuration for the DDIC lookups.

3. The third approach is kind of a mixture of the two above. During development, you persist a repository with all relevant data to a file using the RfcRepository.Save() method. During runtime of the application, the persisted meta-data is restored via RfcRepository.Load() and thus calls to the ABAP system for DDIC lookups are not needed anymore.

The persisted repository can be updated whenever a definition in the backend changes. Thus, the code stays dynamic, but the need of a repository connection is dropped.

Every data container (no matter whether it represents a function, a structure, or a table) provides setter and getter methods for the various different data types, like string, integer, floating point, structure, table, etc.

The big picture of how to use these data container objects now looks as follows. If you are writing an RFC client program, you

- obtain an `IRfcFunction` object (data container for the entire function module at hand) from a repository
- set the necessary importing parameters on the `IRfcFunction` object (this creates necessary tables, structures and substructures in lazy style as they get filled)
- execute the function in the backend (the NCo runtime reads the backend's response from the network connection and populates the `IRfcFunction` object's exporting parameters, again in lazy style)
- read the function module's exporting parameters from the `IRfcFunction` object

If you are writing an RFC server program, you

- start waiting for an incoming call
- once a call arrives, the NCo runtime creates an `IRfcFunction` object for the corresponding function module and populates its importing parameters with the values obtained from the backend
- then NCo passes control to your program, handing over that `IRfcFunction` object.
- you read the importing parameters from that object and fill its exporting parameters as necessary
- when you are done, you simply return control back to the NCo runtime, which reads the exporting parameters from the `IRfcFunction` object and sends them back to the backend system.

The above short example shows that working with NCo 3.1 can be just as simple. All you need is the names of a function module and its parameters, and then with just a few lines of code you can already execute that function module. Also, people having a bit of experience with SAP JCo or the SAP NetWeaver RFC library will easily recognize the similarities between these three solutions, making the learning curve quite steep. No matter whether you are developing in C, Java or C#, you can use the same kind of concepts and patterns.

RFC Client Programs

Now let's put all the pieces together that we learned in chapters [Providing Logon Parameters](#) and [Working with Data Containers and the Repository](#). We start developing the first complete RFC client program. However, before we start, there's one more surprise waiting for you: there aren't any RFC connections in NCo 3.1 that you need to manage yourself!

In NCo 3.1, you work with destination objects. A destination gets all necessary login parameters from the central configuration and the NCo runtime handles all opening, pooling and closing of RFC connections internally. The benefits are, you no longer need to write the same kind of standard code over and over again, the risk of a connection leak caused by your code is lowered, and the slight loss of flexibility should not matter that much.

Here's how all that works: first you write a class that can provide the necessary login information (or you provide that information in the `app.config` file in PoC or demo applications). In our simple example we just hard-code the login parameters, which is of course bad practice. In a productive application, you will make a lookup in a database, some other data storage or prompt the user for input. This example makes use of a simple application server based connection using classic CPIC connectivity. If load balanced connection via message server should be used, or instead a WebSocket RFC based connection should be established, only the set of logon parameters would differ, the rest of the example code remains exactly the same.

```
public class MyBackendConfig : IDestinationConfiguration
{
    public RfcConfigParameters GetParameters(String destinationName)
    {
        if ("PRD_000".Equals(destinationName))
        {
            RfcConfigParameters parms=new RfcConfigParameters();
            parms.Add(RfcConfigParameters.AppServerHost, "127.0.0.1");
            parms.Add(RfcConfigParameters.SystemNumber, "01");
            parms.Add(RfcConfigParameters.User, "MICKEY_MOUSE");
            parms.Add(RfcConfigParameters.Password, "secret");
            parms.Add(RfcConfigParameters.Client, "000");
            parms.Add(RfcConfigParameters.Language, "EN");
            parms.Add(RfcConfigParameters.PoolSize, "5");
            parms.Add(RfcConfigParameters.MaxPoolSize, "10");
            parms.Add(RfcConfigParameters.IdleTimeout, "600");
            return parms;
        }
        else
        {
            return null;
        }
    }

    // The following two are not used in this example:
    public bool ChangeEventsSupported()
    {
        return false;
    }

    public event RfcDestinationManager.ConfigurationChangeHandler ConfigurationChanged;
}
```

Register the above class `MyBackendConfig` with the `RfcDestinationManager`. This is usually done within some startup code.

The next steps are then slightly different, depending on whether we execute a synchronous RFC or a transactional RFC.

Synchronous RFC

In the case of a synchronous RFC, we proceed as follows:

- Get the destination, on which you want to execute RFC calls, from the `RfcDestinationManager`.
- Get a repository from the destination. (Now we finally know, where these repository objects can be obtained from) Normally, for its DDIC lookups the repository will simply use the user specified in the destination. However, you can also specify a different user for the DDIC lookups by adding the configuration parameters `RfcConfigParameters.RepositoryUser` and `RfcConfigParameters.RepositoryPassword` to the `RfcConfigParameters` object for this destination. You can even instruct the repository to use a different backend system for its DDIC lookups by specifying the parameter `RfcConfigParameters.RepositoryDestination`. This can be useful, if you know

that both systems have identical function module definitions and you want to cache this information only once on .NET Connector side. (Normally the .NET Connector maintains one separate DDIC cache for each backend system, in order to make sure that different definitions for the same function module don't lead to data corruption.) So you would setup destination A as usual, and then for destination B you would define `RepositoryDestination="A"`. This makes the two destinations share the same repository and consequently all function module and structure definitions are cached only once.

- Create a function object from the repository.
- Fill the function with the necessary importing parameters as outlined in chapter [Working With Data Containers and the Repository](#).
- Execute the function against the destination.
- Read and process the exporting parameters.

In terms of code, this looks as follows:

```
public static void Main(string[] args)
{
    RfcDestinationManager.RegisterDestinationConfiguration(new MyBackendConfig());
    RfcDestination prd=RfcDestinationManager.GetDestination("PRD_000");

    try
    {
        RfcRepository repo=prd.Repository;
        IRfcFunction companyBapi=repo.CreateFunction("BAPI_COMPANY_GETDETAIL");
        companyBapi.SetValue("COMPANYID", "001000");

        companyBapi.Invoke(prd);

        IRfcStructure detail=companyBapi.GetStructure("COMPANY_DETAIL");
        String companyName=detail.GetString("NAME1");
        Console.WriteLine(companyName);
    }
    catch (RfcCommunicationException e)
    {
        // network problem...
    }
    catch (RfcLogonException e)
    {
        // user could not logon...
    }
    catch (RfcAbapRuntimeException e)
    {
        // serious problem on ABAP system side...
    }
    catch (RfcAbapBaseException e)
    {
        // The function module returned an ABAP exception, an ABAP message
        // or an ABAP class-based exception...
    }
}
```

Stateful RFC Client

In the previous example you may have asked yourself, what is actually happening under the hoods in the line

```
companyBapi.Invoke(prd);
```

Well, the NCo runtime performs the following steps in that single line:

- Fetch an RFC connection from the connection pool corresponding to the destination `prd`.
- Execute the function call over that connection.
- Trigger a reset of the user session state (aka context) on ABAP system side, so that other applications, that may be reusing this connection at a later point in time, won't accidentally run into side effects caused by left-overs of the current function call.
- Return the connection back to the pool, if the destination configuration defines to use a pool. Otherwise, the connection might be closed
- Return control back to the application.

As you can see, this results in a "stateless" call, because the user session state on backend side is always reset after each call. But how can it be achieved, if you need to call an "update BAPI", which requires the user session to be held until the COMMIT WORK has been executed in that user session? Suppose you want to call `BAPI_SALESORDER_CREATEFROMDAT2` followed by `BAPI_TRANSACTION_COMMIT` and you have created two `IRfcFunction` objects for this: `orderCreateBapi` and `commitBapi`. Then two lines like this

```
orderCreateBapi.Invoke(prd);
commitBapi.Invoke(prd);
```

won't work, because there are two problems here: first of all, in a highly loaded application with many concurrent threads, there is a good chance that the second line will pick up a completely different RFC connection from the pool than the first line. In this case, the COMMIT WORK will be triggered inside a completely different user session on backend side. And next, even if the application is lucky and gets the same RFC connection in both lines, the automatic context reset at the end of the first `Invoke()` has already discarded all updates done by `BAPI_SALESORDER_CREATEFROMDAT2`, so the COMMIT WORK comes too late as an implicit ROLLBACK work was done when resetting the context.

For most cases, fixing the above problem is quite easy: all you need to do is let the NCo runtime know that you now want to start a stateful call sequence for the destination `prd`:

```
RfcSessionManager.BeginContext(prd);
orderCreateBapi.Invoke(prd);
commitBapi.Invoke(prd);
RfcSessionManager.EndContext(prd);
```

Simplified, this leads to the following behavior:

- The `BeginContext()` binds an RFC connection from the destinations connection pool to the currently executing thread for exclusive usage.
- The two `Invoke()` calls in lines 2 and 3 first check, whether there is a connection bound to the current thread, and if they find one, they use it for their call. Also the `Invoke()` does not perform a reset of the ABAP session context in that case.
- The `EndContext()` detaches that exclusive RFC connection from the current thread again, performs a context reset on that connection and returns it to the pool.

This way both BAPIs run inside the very same backend user context, and we get the desired result.

Now let's suppose things are a bit more difficult: your .NET application is running inside a web server, and it is processing successive HTTP requests coming from a front-end browser. The HTTP user is logged in to the web server, and its corresponding HTTP user session is identified by a session cookie that is passed back and forth between web server and browser. Let's say the .NET application receives three successive HTTP requests: an order create event, an order change event and in the end the confirmation of the shopping cart. In the first event you want to execute the `SalesOrderCreate` BAPI, in the second one the `SalesOrderChange` BAPI and finally the `TransactionCommit` BAPI. However, the different HTTP requests will normally be processed inside different (random) threads from the web server's thread pool. Therefore, our previous approach will not work here, and a more sophisticated approach needs to be used.

What you need to do in such a case is to create your own `ISessionProvider` object and register it with the `RfcSessionManager`. This allows you to "bind" an exclusive RFC connection to any kind of "ID" (in our case the session ID from the HTTP session cookie) instead of to the current thread. Consequently, you can use this exclusive connection from any number of random threads and have one dedicated user session on ABAP system side. However, you need to be careful not to use this connection from two different threads at the same time. If you still use it concurrently, your application will end up in an `RfcConcurrentCallException` in the thread starting the second `Invoke()`.

An example of how to do this, would be beyond the scope of this overview document, and therefore we refer you to the main documentation or the tutorial, if you need to perform this task. Here just a quick note that for this scenario only the methods `GetCurrentSession()` and `IsAlive(String SessionID)` are required. The other methods of the `ISessionProvider` interface are used or necessary only in an RFC server scenario and High Availability setups and may remain "empty" for simple RFC client scenarios.

One more technical remark for the curious: The default behavior of binding connections to the current thread in the case of `RfcSessionManager.BeginContext(prd)`, is actually achieved via a special `ISessionProvider` implementation that comes with the .NET Connector and is used by default, if the application does not register its own session provider instance.

tRFC/qRFC/bgRFC

For sending a transactional RFC, a few more steps are necessary. We will walk through this once for tRFC. Making a qRFC or bgRFC call is completely analogous: in the following just add a queue name for qRFC or replace the `RfcTransaction` object with an `RfcBackgroundUnit` object for a bgRFC.

- Create an `RfcTransaction` object.
- Get the TID from the transaction and save the data payload together with the TID for example into a database, so you can later repeat the transaction in case it fails the first time. (If this is already the second attempt, you just read the saved data and TID and then create an `RfcTransaction` object with the existing TID.)
- Get the destination, against which you want to execute RFC calls, from the `RfcDestinationManager`.
- Get a repository from the destination.
- From the repository create the function module(s) to be used in the tRFC.
- Fill the payload data into the `IRfcFunction` object(s) as described in *Working With Data Containers*.
- Add the `IRfcFunction` object(s) to the `RfcTransaction` object.
- Execute the `RfcTransaction` object on the destination.
- If the above fails, i.e., it throws an exception, mark that TID for a later retry and store the content of the `RfcTransaction` object for that purpose.
- If the call ends successfully, delete the payload and TID from your database and if that is successful as well, confirm the TID on the destination.

By using the above procedure, you should be able to write a 100% fireproof transactional program, i.e., neither duplicate transactions nor lost transactions can occur.

The coding for this could look as follows:

```
public static void Main(string[] args)
{
    RfcDestinationManager.RegisterDestinationConfiguration(new MyBackendConfig());

    RfcTransaction trans=new RfcTransaction();
    RfcTID tid=trans.Tid;
    String[] payload = ...; // In this example we just send a bunch of strings into
                            // the backend.

    // Save tid.TID and payload into a database.
    RfcDestination prd=RfcDestinationManager.GetDestination("PRD_000");
    try
    {
        RfcRepository repo=prd.Repository;
        IRfcFunction stfcWrite=repo.CreateFunction("STFC_WRITE_TO_TCPIC");
        IRfcTable data=stfcWrite.GetTable("TCPICDAT");

        data.Append(payload.Length);
        for (int i=0; i<payload.Length; ++i)
            data[i].SetValue("LINE", payload[i]);

        trans.AddFunction(stfcWrite);
        trans.Commit(prd);
    }
    catch (Exception e)
    {
        // Log the error, so that an admin can look at it, fix it and retry
        // the transaction.
        return;
    }

    // We should execute the Confirm step in the backend only after we are 100%
    // sure that the data has been deleted on our side, otherwise we risk a
    // duplicate transaction!
    try
    {
        // Delete tid and payload from database.
    }
    catch (Exception e)
    {
        // No confirm in this case. The data may still be there and may
        // accidentally get resent automatically at a later time. The backend
        // still needs to protect against this possibility by keeping the TID.
        return;
    }

    prd.ConfirmTransactionID(tid); // This deletes the tid from ARFCRSTATE on
                                  // backend side.
}
```

After executing this program, you should log on to the backend system, go to transaction *SE16* and display the contents of the database table *TCPIC*. You should be able to find your strings in the column named "BUFFER".

RFC Server Programs

The initial steps of server programming are similar to those we have seen for client programming: first you create a class that provides the necessary configuration parameters for the gateway where your RFC server needs to register. (Alternatively, you can define them in the *app.config* file of your application.) Again, we will use a very simple hard-coded example for this, which is actually not to be recommended for productive usage. And again, this is only an example for a simple registered server based on CPIC. If choosing a different server type like the WebSocket RFC server, the parameters to be used will differ.

```
public class MyServerConfig : IServerConfiguration
{
    public RfcConfigParameters GetParameters(String serverName)
    {
        if ("PRD_REG_SERVER".Equals(serverName))
        {
            RfcConfigParameters parms=new RfcConfigParameters();
            parms.Add(RfcConfigParameters.GatewayHost, "127.0.0.1");
            parms.Add(RfcConfigParameters.GatewayService, "sapgw01");
            parms.Add(RfcConfigParameters.ProgramID, "DOT_NET_SERVER");
            parms.Add(RfcConfigParameters.RepositoryDestination,"PRD_000");
            parms.Add(RfcConfigParameters.RegistrationCount, "5");
            return parms;
        }
        else
        {
            return null;
        }
    }

    // The following two are not used in this example:
    public bool ChangeEventsSupported()
    {
        return false;
    }

    public event RfcServerManager.ConfigurationChangeHandler ConfigurationChanged;
}
```

Note, how we simply tell the RFC server to use the destination "PRD_000" for its DDIC lookups. ("PRD_000" was used in chapter [RFC Client Programs](#), please revisit that if necessary.)

Alternatively, we could also create a server without repository destination and then set a repository on the server object, before we start the server. This could then be for example a hard-coded repository that we created ourselves.

An object of the above *MyServerConfig* class needs to be registered with the *RfcServerManager* during startup of your application.

The server programming in NCo 3.1 is quite simple: there is only one generic server class that is handling the incoming requests. Likewise, the generic data container classes from chapter [Working with Data Containers and the Repository](#) are used for accessing data received for a concrete function module request.

In order to properly process the incoming function module request, we need to implement one or more methods. For this, there are several possibilities:

- One static method for each function module. You assign a method to a function module by adorning the method with an attribute as follows:

```
[RfcServerFunction(Name="STFC_CONNECTION", Default=false)]  
public static void Function(RfcServerContext context, IRfcFunction function)
```
- One static method for all function modules (or for all function modules that don't have an explicit method assigned to it. Explicit implementations are preferred to the default implementation). This is called the "default" method. Use the same attribute as above and set the `Default` parameter to `true`. The `Name` parameter is irrelevant in this case and can be omitted.
- One instance method for each function module. Use the same attribute as in the first case.
- One instance method for all function modules (or for all function modules that don't have an explicit method assigned to it. Explicit implementations are preferred to the default implementation). Use the same attribute as in the second case.

What is the difference between using static methods and instance methods? Static methods are generally used for stateless RFC servers, meaning the function module does not need to preserve state information between different invocations. If you use instance methods, the NCo runtime creates a new object instance of the surrounding class for each user session on ABAP system side and keeps it as long as that user session (and thus the RFC connection between that user session and the .NET program) is alive. That way the .NET application can use the instance variables of that object for keeping state between several calls of the function module. You can even keep and use state information between calls of different function modules, if the implementing methods for these function modules are provided by the same object. However, if you don't like to have that many objects getting created, you can just as well implement your function modules as static methods and keep any state information in a dedicated session related storage.

We will describe the details for implementing a stateful server in chapter [Stateful RFC Server](#), but first let's take a look at a simple stateless server that receives synchronous RFC calls.

Synchronous RFC

The general steps for starting a simple RFC server are as follows:

- Implement a handler class (usually with static methods) as described above.
- Create an `RfcServer` object using the specified configuration parameters and the type of your handler class. (You may provide multiple handler classes. The NCo runtime will inspect these classes, and all methods that are annotated with the `RfcServerFunction` attribute, will be used for processing incoming function calls.)
- Start the `RfcServer` object.

Sample code for this could look as follows:

```
public class MyServerHandler
{
    [RfcServerFunction(Name="STFC_CONNECTION")]
    public static void StfcConnection(RfcServerContext context, IRfcFunction function)
    {
        Console.WriteLine("Received function call {0} from system {1}.",
            function.Metadata.Name, context.SystemAttributes.SystemID);

        String reqtext=function.GetString("REQUTEXT");
        Console.WriteLine("REQUTEXT = {0}\n", reqtext);

        function.SetValue("ECHOTEXT", reqtext);
        function.SetValue("RESPTTEXT", "Hello from NCo 3.1!");
    }
}

public static void Main(String[] args)
{
    RfcDestinationManager.RegisterDestinationConfiguration(new MyBackendConfig());
    RfcServerManager.RegisterServerConfiguration(new MyServerConfig());

    Type[] handlers=new Type[1] { typeof(MyServerHandler) };
    RfcServer server=RfcServerManager.GetServer("PRD_REG_SERVER", handlers);

    server.Start();
    Console.WriteLine("Server has been started. Press X to exit.\n");

    while (true)
    {
        if (Console.ReadLine().Equals("X"))
            break;
    }
    server.Shutdown(true); //Shuts down immediately, aborting ongoing requests.
}
```

In case you haven't noticed: this simple program can serve up to 5 backend work processes simultaneously in 5 parallel threads. (This is why we set `RegistrationCount` to 5 in the `MyServerConfig` class.)

There are a few more features worth mentioning here. An `RfcServer` object provides three events, at which you can register an event handler, if you are interested in getting notifications of various error or state-change situations:

- **RfcServerError**
This event is triggered, whenever a low-level error happens inside the `RfcServer`, which is outside of the control of the application. For example, network problems between the server program and the backend.
- **RfcServerApplicationError**
This event is triggered, whenever one of the `RfcServerFunctions` (like the `StfcConnection` function in the sample code) throws an exception. Normally, this exception should be nothing new to your application, but in case you want to implement some logging, it may be easier to do it in one central place rather than to have to do it in every single `RfcServerFunction`.
- **RfcServerStateChanged**
This event is triggered, whenever the server's connection state changes, e.g., it temporarily goes into state "Broken", because the connectivity to the backend is currently down.

Another interesting aspect is the topic of reporting error situations back to the ABAP system. The two commonly used cases are the classic ABAP Exception (for all application/business level problems, like: the backend system is requesting information about a customer ID xy, but the .NET application does not know any customer with ID xy) and the classic `SYSTEM_FAILURE` (for all technical problems happening in the `RfcServerFunction`, like an out of memory situation or a broken connection to a database while looking up information from that DB).

You can send an ABAP Exception back to the backend simply by throwing `RfcAbapException` from your `RfcServerFunction`, and you can send a `SYSTEM_FAILURE` by throwing `RfcAbapRuntimeException`. In addition to that you can also use ABAP Messages (`RfcAbapMessageException`).

The final point of interest for all kinds of RFC server programs may be "user authentication and user authorizations". You may not want to allow every backend user to execute all the functions in your server program. If you want complete control over which backend user is allowed to execute which function module implementation on your server, you need to implement the interface `IServerSecurityHandler` and register it via

```
server.SecurityHandler = new MySecurityHandler();
```

before starting the server.

In this security handler object, you can then control, which users are allowed to logon to your external server program, and which user may execute which function modules. Backend users can be identified either by plain-text username, by SSO2/authentication ticket or by X.509 certificates, depending on which kind of authentication has been defined in the settings of the corresponding RFC destination in SM59.

Stateful RFC Server

If you want to write a stateful RFC server, there's some more work to do. Basically, there are two different approaches:

Your .NET program does not have its own user session concept

In this case it is probably best to use the instance method approach and use the function handler object for storing state information for as long as the backend user session is still alive. The .NET Connector runtime will handle the necessary session management for you.

Let's modify the previous example from chapter [Synchronous RFC](#) a bit to turn it into a stateful RFC server:

```
public class MyServerHandler
{
    private int callCount=0;

    [RfcServerFunction(Name = "STFC_CONNECTION", Default = false)]
    public void StfcConnection(RfcServerContext context, IRfcFunction function)
    {
        Console.WriteLine("Received function call {0} from system {1}.",
            function.Metadata.Name, context.SystemAttributes.SystemID);

        String reqtext = function.GetString("REQUTEXT");
        Console.WriteLine("REQUTEXT = {0}\n", reqtext);

        ++callCount;

        function.SetValue("ECHOTEXT", reqtext);
        function.SetValue("RESPTXT", "You have called this function "+
            callCount.ToString()+" times so far");

        if (!context.Stateful)
            context.SetStateful(true);
    }
}
```

There are three differences here: we introduced a variable for storing information between different invocations of the function module, we made the implementing method an instance method and most importantly we introduced the line

```
context.SetStateful(true);
```

Without this line the NCo runtime would close the RFC connection to the backend and that way end the session.

Now call this program from *Function Builder (SE37)* several times (by executing STFC_CONNECTION against the RFC destination DOT_NET_SERVER) and you will notice how the counter keeps increasing. Then execute a "/n" in the OK-code field in your SAPGui session. This ends your current ABAP user session and starts a new one. When you now go to *Function Builder* and execute the function module again, you will notice that the counter starts at 1 again, so a new instance of the MyServerHandler class has been created on .NET side.

A simple test of how NCo handles several parallel backend user sessions is the following: start a second SAPGui session (for instance by executing `"/ose37"` in the ok-code field) and then keep calling `STFC_CONNECTION` first a couple of times from one SAPGui session, then from the other one. You will notice how two different counters are getting incremented, depending on which SAPGui session is being used. This means that the NCo runtime has created two different instances of the `MyServerHandler` class and is using the associated one for each of the two SAPGui sessions.

Your .NET program has its own user session concept

In this case you need a "bridge" between the ABAP system's user session management and the user session management of your .NET program. Whenever an ABAP backend user "logs in" to your server application, a new user session of your .NET application needs to be created and associated with the ABAP backend user session. In order to achieve this, you need to implement the interface `ISessionProvider` and register an object of that type with the `RfcSessionManager`.

This is similar to what we did in chapter [Stateful RFC Client](#) for the advanced stateful RFC client scenario. The difference is only that now you need to implement the remaining methods of the `ISessionProvider` interface. The .NET Connector runtime will then interact with the session management of your application in the following way:

```
public String CreateSession()
```

Whenever the ABAP backend system opens a new connection to the external server program, the NCo runtime calls this method and requests your application's session management to create a new user session and attach the current thread to that new session.

Note:

This is partly related to `IServerSecurityHandler`. However, we decided to keep the two notions of "session management" on one side and "user logon check" on the other side separated from each other, so that those applications that are not interested in session management and stateful communication, can still protect their server application by logon and user validation mechanisms. See the interface `IServerSecurityHandler` for more details.

```
public void PassivateSession(String sessionID)
```

The .NET Connector calls this method, whenever an RFC request, which is processed as part of a stateful server connection, has been finished and the corresponding stateful server connection is now idle (waiting for the next RFC request from the backend). The `ISessionProvider` should detach the given user session from the current thread.

```
public void ActivateSession(String sessionID)
```

The .NET Connector calls this method, whenever a new RFC request comes in over an already existing stateful server connection. The `ISessionProvider` should attach the given user session to the current thread.

```
public void DestroySession(String sessionID)
```

The .NET Connector calls this method, whenever the ABAP backend system (or the implementation of a server function) closes a stateful server connection. The application's session management framework should now delete the corresponding user context.

```
public bool IsAlive(String sessionID)
```

Allows the .NET Connector to test, whether a particular application session is still alive.

Once you have implemented an object like this, all you need to do to make your RFC servers interact with your application's session management framework is to add a line like

```
RfcSessionManager.RegisterSessionProvider(new MySessionProvider());
```

to the startup code in the `Main()` method of the previous example.

tRFC/qRFC/bgRFC

We will illustrate the transactional RFC types with a tRFC example again. As in client side tRFC processing, it is important that you have a database or other kind of transaction store, where you can keep status information of ongoing transactions securely and permanently, and that you respond correctly to the various events that occur during tRFC processing.

The first tasks for implementing a tRFC server are the same as for every RFC server: you need to implement the necessary handler function(s) for the function module(s) contained in the tRFC LUWs that you want to receive. Please note that one tRFC LUW may contain several function module calls. These are then supposed to be processed one after the other, and they are considered as one single atomic unit, i.e., they should be committed all at once, or not at all. (However, in 99% of the cases a tRFC LUW contains only one single function call.)

Next you need to implement the interface `ITransactionIDHandler`. That class will interact with your status management component and that way guarantee transactional security (execution "exactly once") of the LUWs your application will receive. In the four methods of that class, you need to do the following:

- **CheckTransactionID**
A new tRFC/qRFC LUW arrived from the backend and we need to check, whether the corresponding TID is already known on our server. The Check-function should now search the status database for the given TID. This search can lead to the following three results:
 - The connection to the database is currently down, or some other internal error happens. In this case just throw an `RfcInternalError`. The .NET Connector will then send a `SYSTEM_FAILURE` back to the ABAP system, and the ABAP system can retry the same transaction sometime later.
 - The TID does not yet exist in the TID database, or it exists and has the status `Created` or `Rolled back`. In this case, the Check-function should create an entry for this TID (if not already existent), set the Status to `Created` and return "true". The .NET Connector will then proceed executing the function modules contained in the tRFC/qRFC LUW, i.e., it will call the function handlers for the corresponding function module names.
 - The TID already exists in the TID database and has the status `Committed`. The Check-function should simply return "false" in that case. The .NET Connector will then return an OK message to the ABAP system without executing the LUW a second time. The ABAP system then knows that the transaction has already been processed successfully and can continue with the `ConfirmTID` step.
- **Commit**
After all function modules contained in the current LUW have been executed successfully, the .NET Connector calls this method. The application should persist all changes done for this TID in the previous function module handler functions. If this is passing successfully, it should set the status to `Committed`.
- **Rollback**
If one of the function modules contained in the current LUW ended with an exception, the .NET Connector calls this API. It should rollback all changes done for this TID in the previous function modules. Afterwards, it should set the status to `Rolled back`.

- `ConfirmTransactionID`
When the backend system finds out that the LUW corresponding to this TID has been executed completely and successfully on the RFC server side, it will trigger a `ConfirmTID` event. When receiving this event, the .NET Connector will call this API. It should simply delete the entry for this TID from the database to clean up resources.

A fully functional example of this class would certainly go beyond the scope of this overview article. For more details see the full NCo documentation. Once you have implemented a class that does all the necessary work (e.g., `MyTIDHandler`), you enable your RFC server for tRFC/qRFC processing by simply adding a line like

```
server.TransactionIDHandler = new MyTIDHandler();
```

before the `server.Start()` call.

Enabling an RFC server for processing bgRFC is similar. The only difference is that instead of the `ITransactionIDHandler` interface, you need to implement the interface `IUnitIDHandler` and assign an instance to your server.