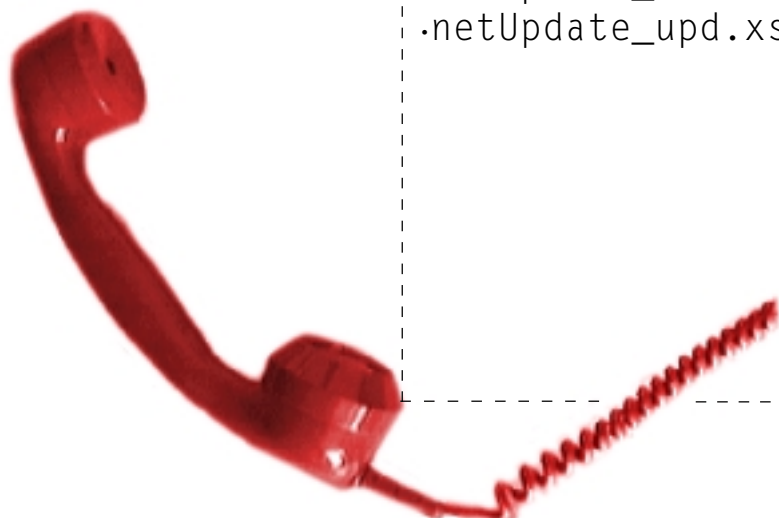




reference manual

# *netUpdate project*

an overview of the process and a description  
of the involved XSD schema files



- netUpdate\_def.xsd
- netUpdate\_upd.xsd



TMR AG  
Triangle Micro Research  
©TMR AG - all rights reserved

TMR - Triangle Micro Research AG  
Ribigasse 3-5, CH-4434 Hölstein

tel: (061) 956 99 00  
fax: (061) 956 99 10  
mail: [info@tmr.ch](mailto:info@tmr.ch)  
web: <http://www.tmr.ch>



## Table of contents

1.	Introduction .....	2
2.	Anatomy of a netUpdate process .....	3
2.1	Components involved in a netUpdate process .....	3
2.2	State diagram of the update process .....	4
2.3	Conditions for an update .....	6
2.4	Chicken and eggs .....	6
2.5	Declaration of dependencies among netUpdate projects .....	7
2.6	Parameters in the definition and update file .....	7
2.7	How to prohibit the update of a project .....	8
2.8	Incremental update of a resource .....	8
2.9	Postprocessing after the update of a netUpdate project .....	9
2.10	What happens after the loss of a net connection .....	9
3.	Security .....	10
4.	Reference documentation of the schema definitions .....	11
4.1	Conventions, terms, and definitions used .....	11
4.2	netUpdate_def.xsd – netUpdate definition schema .....	12
4.3	netUpdate_upd.xsd – netUpdate update schema .....	14
5.	Version history .....	21

Permission to use, and distribute this documentation and its accompanying XSD schema files for any purpose and without fee is hereby granted in perpetuity, provided that the copyright notice (© 2001 by TMR AG) and this paragraph appear in all copies and neither the documentation nor the schema files are altered in any way.

The copyright holders make no representation about the suitability of the schema for any purpose. It is provided "as is" without expressed or implied warranty.

## 1. Introduction

This document gives an overview about the netUpdate project, its purpose and principle workflow of the processes taking place. On the other hand it serves as reference manual for the XSD schema file used within the netUpdate project. The focused XSD schema files are

- *netUpdate\_def.xsd* XSD schema definition for a netUpdate definition file
- *netUpdate\_upd.xsd* XSD schema definition for a netUpdate update file

The target audience of this documentation are software developers who like to get an understanding of the processes in the netUpdate environment or need to provide the necessary files for a netUpdate project. For the understanding of the following we presuppose a rudimentary understanding of the XML and XSD standards, respectively.

The current version and all upcoming updates of this documentation are available online at the URL <http://www.netupdate.ch/en/netUpdate/>. The current versions of the XSD schema files are available online at the URL <http://www.netupdate.ch/en/netUpdate/>. Note that this documentation is not intended as user manual for the netUpdate application.

## 2. Anatomy of a netUpdate process

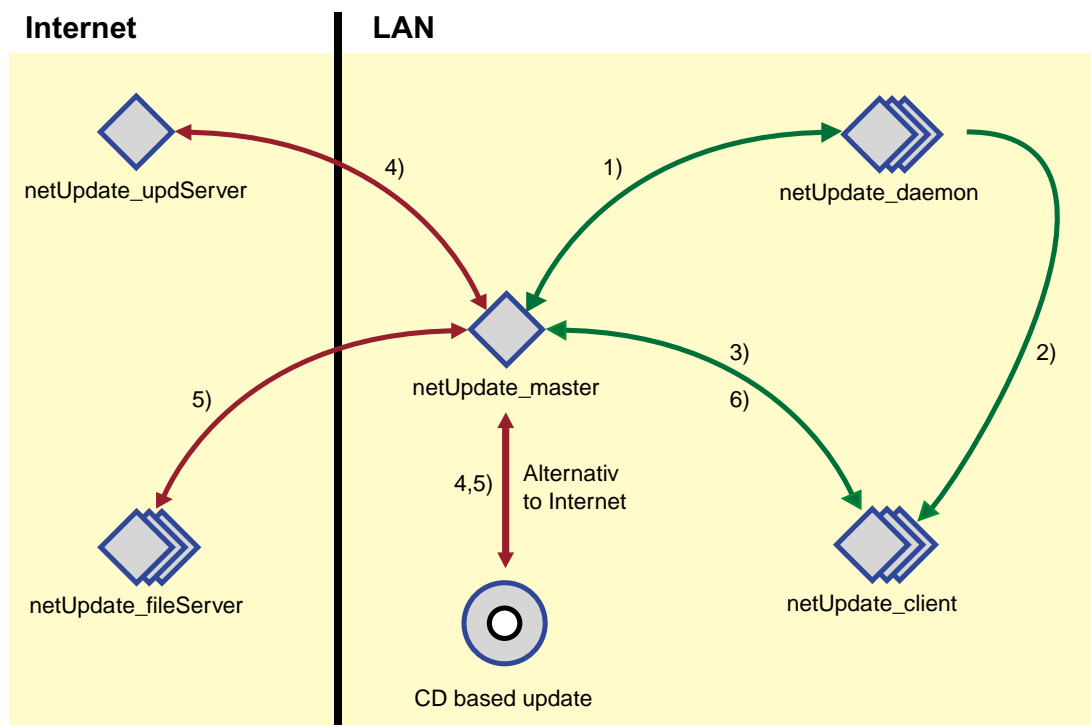
The following paragraphs give an introduction into the naming conventions and components used, the process diagram of a netUpdate process, and the conditions for an update. Furthermore various principal aspects like the declaration of dependencies, the usage of parameters etc. are highlighted.

### 2.1 Components involved in a netUpdate process

The following table names and describes the components of the netUpdate package that are used in a netUpdate process.

name	description
netUpdate_master	Is the master process run in the LAN having access to the internet and supplies the user with all necessary information via a GUI. In principle all LAN hosts with an access to the internet can run the netUpdate_master. The user primarily start the netUpdate_master to begin with an update process
netUpdate_client	Each host that needs to be updated has a netUpdate_client installed. The netUpdate_client is started automatically upon request by the netUpdate_daemon.
netUpdate_daemon	Is a very small footprint daemon process started at system startup and listening to requests of the netUpdate_master. Once such a request is broadcasted over the LAN by the netUpdate_master, the netUpdate_daemon starts the netUpdate_client in turn. The later one then handles all communication for that session.
netUpdate_updServer	A web server with a public URL address sending back the netUpdate .upd file upon request. The URL of that server is defined on a per project basis in the netUpdate .def file.
netUpdate_fileServer	A web server with a public URL address sending back requested files. The URL of that server is defined on a per file basis in the netUpdate update file.

The following figure gives a rough overview about the interaction of these components forming as a whole the netUpdate process.



To get these interactions to work there are 2 important and mandatory data sources necessary: the netUpdate definition file and the netUpdate update file, respectively. Both of these files are XML data files for which the XSD schema files are given in the next chapter.

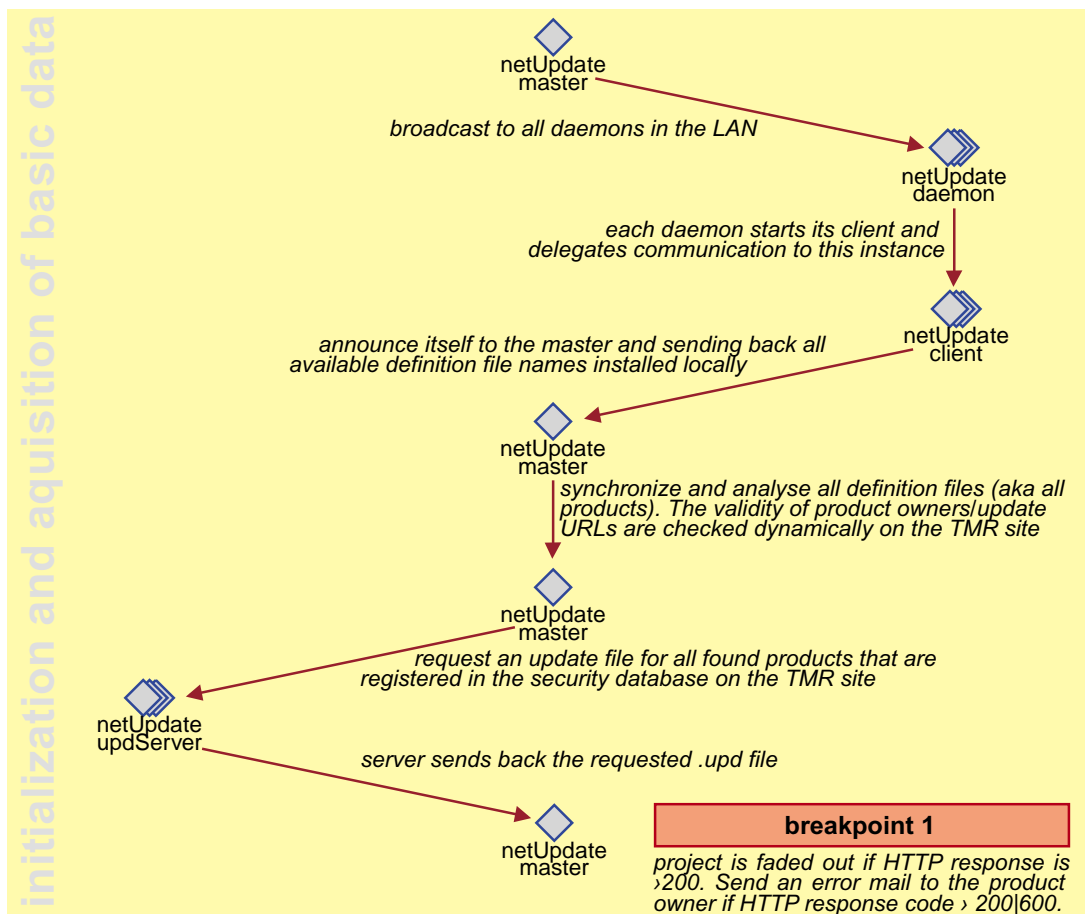
The next table summarises these files:

name	description
definition file (extension .def)	The netUpdate definition file is stored in the data directory of the netUpdate root and supplies all necessary information for one product to acquire the update information from the netUpdate_updServer (step 4). This file is as well inside the netUpdate space of that product and therefore updateable (cf. paragraph "chicken and eggs"). While not mandatory it is recommended to name this file after the product it represents (maybe with a language selector) and to give it an extension .def. This file must be in conformity with the netUpdate_def.xsd schema file!
update file (extension .upd)	The netUpdate update file, acquired always directly from the netUpdate_updServer, holds all information about the resources belonging to the focused product. It is the necessary source for making smart update decisions and requesting the needed resources from the netUpdate_fileServer (step 5). While not mandatory it is recommended to name this file after the product it represents (maybe with a language selector) and to give it an extension .upd. This file must be in conformity with the netUpdate_upd.xsd schema file!

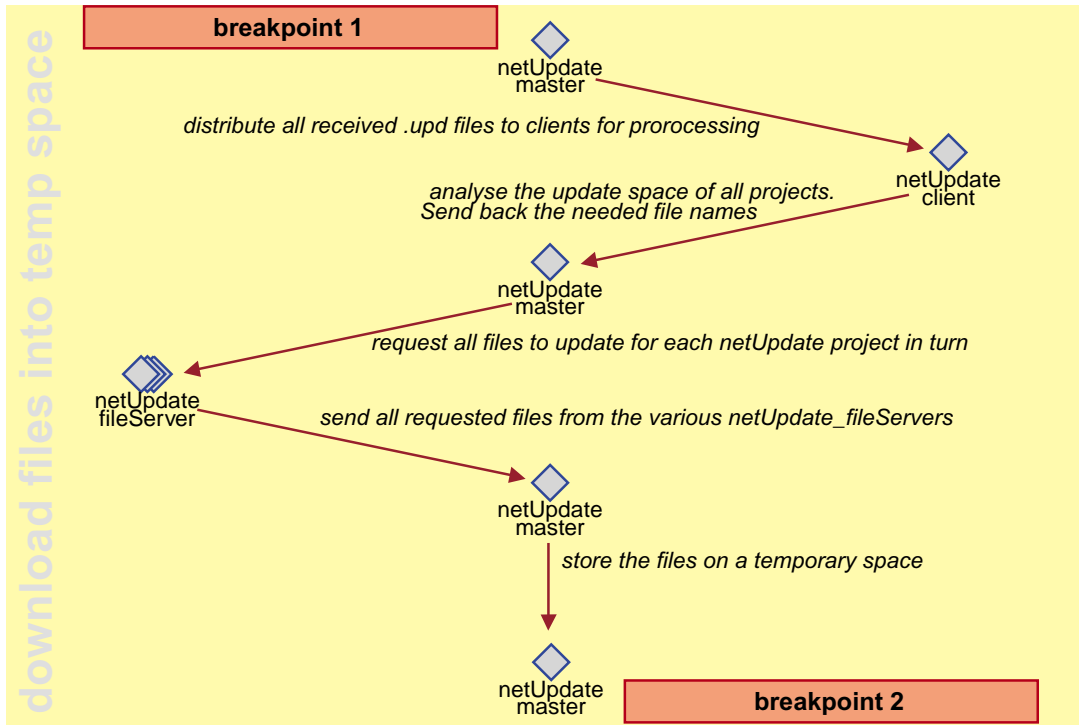
### 2.2 State diagram of the update process

The state diagram of the netUpdate process can be visualised as reaching a series of breakpoints and some actions the netUpdate\_master has to process moving from one to next breakpoint. Per definition it is safe to stop or restart the netUpdate process at these breakpoints (cf. paragraph "net connection loss").

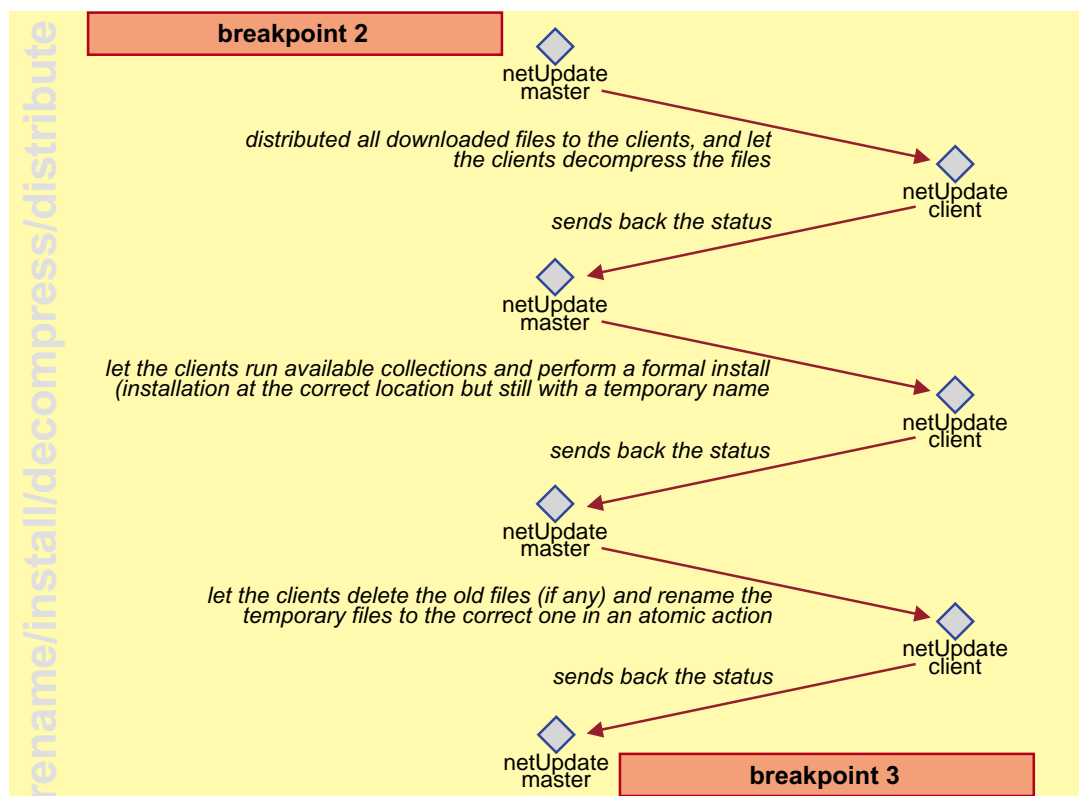
The following sequence of graphs shows and describes all principle actions taking place between 2 breakpoints. Since the update information is always handled dynamically the first step (up to breakpoint 1) is always carried out regardless of the state of the process (a prior net connection interrupt or a new update process).



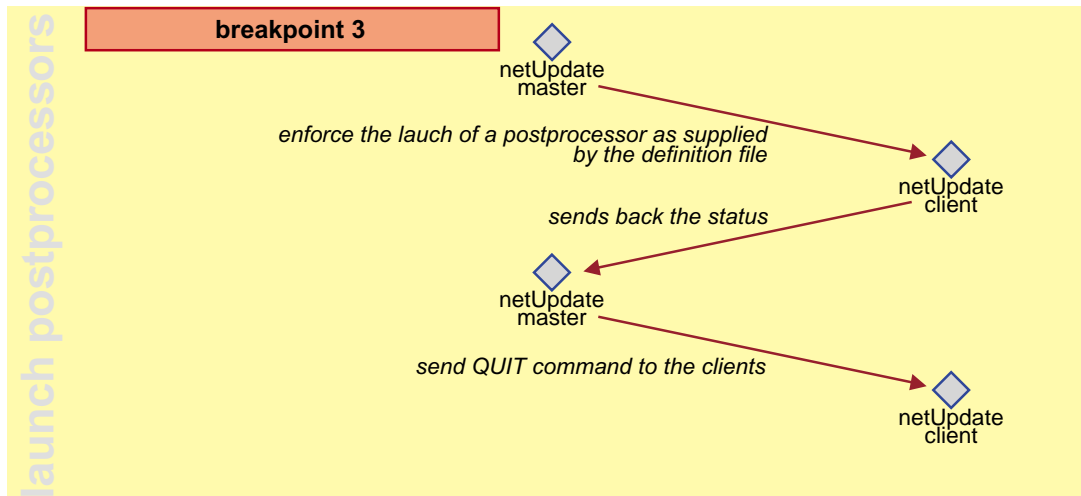
In the next step up to breakpoint 2 all netUpdate projects are analysed and the set of files to update will be downloaded onto a temporary space, which is under the management of the netUpdate\_master only. Please note that a special situation arises if the netUpdate project itself needs an update. Under that condition the netUpdate\_master only requests files from that project and discards any other projects. The whole process is done the same way but only with the netUpdate project as target, other projects are faded out. Therefore a new update process should be started after the initial update to ensure that all other projects are up-to-date as well.



From breakpoint 2 on the “net communication” is considered stable, fast and reliable, as it is network traffic solemnly in the LAN. Therefore the whole distribution, decompressing, handling of available collections, and installation is taking place in the next step.



The last step in a netUpdate process is the execution of any postprocessor directive (as defined in the definition file of a project). Note that the launch sequence of the postprocessor is arbitrary.



### 2.3 Conditions for an update

Each netUpdate project consists of a set of resource definitions. A resource definition in turn is defined basically by a remote URL to fetch the resource from, a local absolute file path, an optional installation flag, and a modification date of the remote resource.

There are 7 conditions for which an update might take place:

- 1) an unconditional install is performed if the installation flag is set to "mustInstallWithPathGeneration" if and only if the resource does not exist locally. The directory path will be generated if it does not already exist,
- 2) an unconditional install is performed if the installation flag is set to "mustInstallIfPathExist" if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install,
- 3) an unconditional update is performed if the installation flag is set to "mustUpdateWithPathGeneration" even if the resource or the directory path does not exist (the path will be generated if it does not already exist),
- 4) an unconditional update is performed if the installation flag is set to "mustUpdateIfPathExist" even if the resource does not exist locally. However, the directory path must already exist to perform the forced update,
- 5) an unconditional install is performed if the installation flag is set to "mightInstallWithPathGeneration" if and only if the resource does not exist locally. The directory path will be generated if it does not already exist. In the case that the resource does exist, then a conditional update is performed under the same conditions as with "mightUpdate",
- 6) an unconditional install is performed if the installation flag is set to "mightInstallIfPathExist" if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install. In the case that the resource does exist, then a conditional update is performed under the same conditions as with "mightUpdate",
- 7) a conditional update is performed if the installation flag is set to "mightUpdate" (the default value) if and only if the corresponding resource exists locally and the last modified date of that local resource is less than the remote modification date.

Once an update is performed on a resource, the creation and modification date is set to the remote modification date plus one hour and one second (be aware of day light savings time differences!). This assures that no further update is taking place the next time a netUpdate process is started on that project unless the netUpdate update file was changed again to reflect a new "update state".

### 2.4 Chicken and eggs

Since the netUpdate environment can update itself there is a kind of "chicken and egg" problem. It is therefore imperative to understand the exact steps and actions taken to solve that problem. As explained in the state diagram the netUpdate project is updated prior to any other project. If any of the resources of the netUpdate project must be updated then all other projects are discarded in that run and the process is stopped after the download and installation phase are done. To check for "normal" updates the user has to restart the netUpdate application again.

In the second attempt to update netUpdate projects (if the process was stopped in the initial attempt) it is very likely that the netUpdate environment and any definition files are already up-to-date. The "normal" update process can therefore take place. As a side effect of stopping the initial update the new netUpdate environment are used from the very beginning.

## 2.5 Declaration of dependencies among netUpdate projects

In the world of components (CORBA, JavaBeans, etc.) there are explicit and implicit dependencies between producer and consumer. It is essential that the versions installed can coexist with each other or that a requested producer component is available at all. Under these conditions there is not only the requirement to update all files of a software package (the resources of a project in our language!), but there is as well the necessity to concurrently update other software packages (aka netUpdate projects), that the primary project relies on.

In the update file of a netUpdate project one can define any number of declarations of parent-child dependencies. Since only the developer of the consumer components knows which producer components are used inside the software only forward declarations are acceptable. Of course, dependencies are bi-directional in its interaction. To overcome this problem the netUpdate application analyses all downloaded update files. The knowledge of all forward dependencies allows building a fully connected graph. As a consequence the user can only select/deselect an update of fully independent root leaf projects, whereas the depended components inherit the behaviour (update/no update) of the parents.

To define a parent-child dependency one defines an `<uses ../>` element in the definition file supplying the name of another netUpdate definition file (cf. chapter "Reference documentation of the schema definitions"). Please note that dependencies are declared on a per project basis (aka software package basis) even if only one single file of a project might have such a dependency.

## 2.6 Parameters in the definition and update file

There is a mechanism built into the netUpdate system that allows one to define local absolute file paths using previously defined parameters. These parameters will be expanded in local path names by the netUpdate application. It is essential to note that these parameters are local in scope, that is the expansion is adaptable to the local host environment.

The syntax for using parameters is `${YOUR_PARAMETER}`. All parameter names must be uppercase and begin with a character [A-Z] followed by any number of characters [A-Z] and/or numbers [0-9] and/or the characters underscore and dash [\_-]. Parameters are defined in the `parameter` section of the netUpdate definition and/or update file. The left-hand side in a parameter definition (`value` attribute in our schema) can consist of any number of other parameters defined previously(!) and/or literals and/or fully qualified registry keys. As stated above parameters and registry keys are supplied in the form of `${PARAMETER_NAME}` or `${REG_KEYNAME}` and can be nested as well.

Please note that all parameter values even the fully qualified registry keys are defined in an Unix-like notation using slashes (/)! Furthermore it is assumed that the last character in a directory definition is a /.

```
E.g.  REG  ${HKEY_LOCAL_MACHINE/SOFTWARE/TMR/NETUPDATE/NETUPDATE_DATA}
      ROOT C:/test/sample/
      BIN  ${ROOT}data/
```

The primary use for parameters in the definition file is to transport user localized license information to the netUpdate\_updServer. To submit such a licence piece, a parameter e.g. UID is defined in the definition file pointing to an appropriate registry key. Then one would define the URL of the update file using a GET syntax in the attribute `product:url` of the update file. That is you append `UID=${UID}` as an argument after the end of the base URL separating the two with the question mark character ?, e.g. the URL

```
http://www.netupdate.ch/cgi-bin/netUpdate.cgi?UID=${UID}
```

is expanded with the UID number before sending the request.

Of course, it is possible to send more than just the UID parameter by using the normal GET syntax. The separator between any 2 argument pairs is the ampersand character &. But since the ampersand is a predefined character in an XSD schema files as well the separator must be escaped using the entity `&amp;`. For example the following URL is a legal `product:url` attribute using 3 parameters:

```
http://www.netupdate.ch/getUPD.cgi?product=lab&UID=${UID}&language=fr
```

Before sending the request it is expanded and transformed to the correct HTML syntax

```
http://www.netupdate.ch/getUPD.cgi?product=lab&UID=7612345678910&language=fr
```

### Predefined parameters

The following parameter is predefined by the netUpdate application and can be used to define local file names in an update file only.



parameter symbol	description
NETUPDATE_DATA	The data directory of the netUpdate application where all the definition files reside. This parameter is local in scope, e.g. local to the local netUpdate application. Note that this directory path is available in the local registry as well (given that the netUpdate project was installed) under the key: HKEY_LOCAL_MACHINE/SOFTWARE/TMR/NETUPDATE/NETUPDATE_DATA

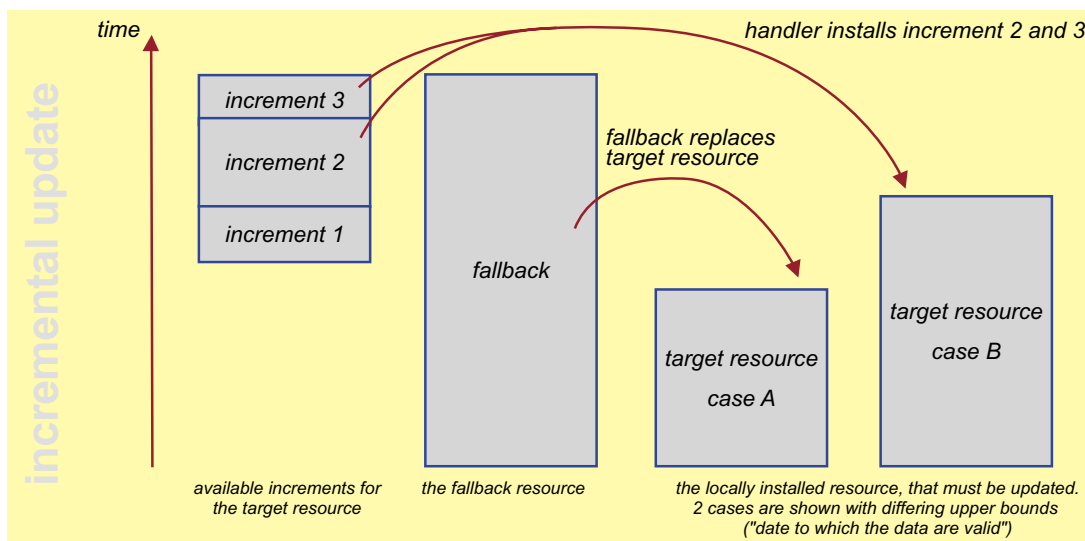
**2.7 How to prohibit the update of a project**

If for any reason the update of a project should be prohibited at a customer’s site, then the target upd-server should send back a HTTP response code 600 – “user has no license”. The project is thereafter not considered in the update space analysis. It’s as if the project is faded out from the available list of projects seen from the netUpdate\_master. Of course, all dependencies and post processor directives are ignored as well under this condition.

**2.8 Incremental update of a resource**

There are resources which change frequently over time but each change is relatively small in size compared to the whole resource, e.g. database files are often of this kind. Therefore, it is desirable to being able to perform a variable number of incremental updates to a target resource. The <collection ../> tackles exactly that increment update scenario. The collection is a container element consisting of an unbounded number of increment resource definitions and one fallback resource definition. For the sake of applying the increment to the target data resource a handler application is defined as well inside the collection.

A prototypical situation in terms of the validity of the data is shown in the following figure. There are 3 incremental files that cover a certain periode in time. The fallback resource –per definition– covers the full date range.



Suppose now that “target resource case A” is installed locally. The upper bound of the target resource (“the date to which the data are valid”) is lower than the increment 1 resource. Even if all increments would be applied to the target resource, the resulting data are still not up-to-date! The netUpdate application will therefore use the fallback to update the target resource meaning that the target resource is replaced by the fallback resource. Note that under this condition the handler application is not launched (because there is no need to do that)!

The situation becomes different if “target resource case B” is installed locally. The upper bound of the target resource falls between the upper bounds of increment 1 and increment 2. The netUpdate application will therefore download increment 2 and increment 3 and the handler application. In the next step the handler application is launched with both increments as arguments. It is now in the responsibility of the handler application to update the target resource using the downloaded increments. Technically speaking the net bandwidth utilization should dramatically decrease compared to case A using the increments otherwise the net result must be the same as if the fallback resource would be installed.

Note that this increment update scheme is based on some assumptions:

- there must always be a fallback solution that is an ordinary resource covering the whole date range,
- if an increment is defined then there must be a defined handler as well.
- a handler is only launched, if at least one increment is downloaded
- the validity date range of an increment is defined by the explicitly given last modified date as the upper date boundary and implicitly by the upper date boundary of the (n-1).th increment. That is the lower date boundary of the n.th increment is the same as the upper date boundary of the (n-1).th increment. This means that the 1<sup>st</sup> increment has no implicit lower date boundary. To overcome this problem, one can define the optional attribute `increment_basedate` in the collection element, to explicitly set lower date boundary of the first element. If the `increment_basedate` is not given then it is assumed, that the first increment spans an date interval of zero.

## **2.9 Postprocessing after the update of a netUpdate project**

It might be desirable to perform some postprocessing after the successful installation of a netUpdate project (or software package that is). Furthermore it even might occur situations where a consumer component want to run a postprocessor on the resources of another consumer project. The `<uses ../>` declaration in the update file allows to define such behaviour by including postprocessor elements. When setting such a declaration one explicitly defines a dependency with another netUpdate project (cf. paragraph “Declaration of dependencies”). To run a postprocessor on the own project, simple define a `<uses ../>` element pointing to the own definition file. The definition of a post processor element is based on the schema definition of an ordinary resource.

All files connected to the focused root leaf product in the dependency graph must be analysed for update. The resulting set of resources must be downloaded, unpacked, and installed successfully before the netUpdate application starts any postprocessors. Furthermore a postprocessor is only started if at least one resource of the target project of that postprocessor is updated. There is no return status check by the netUpdate application. If more then one postprocessor acts on a netUpdate project the launch sequence of these postprocessors is random.

## **2.10 What happens after the loss of a net connection**

Since the Internet connections of most users must be though of unreliable and with low bandwidth, the netUpdate process need to optimise the network utilisation. No installation is taking place before all resources of all analysed projects are downloaded to the LAN successfully. The individual resources are stored on a temporary space on the netUpdate\_master host. Once breakpoint 2 is reached the remaining distribution and unpacking tasks inside the LAN is though of reliable and fast. If all resources are distributed to all netUpdate\_clients, then the temporary space on the netUpdate\_master is purged. From breakpoint 3 on the final installation of the resources is targeted and finally the postprocessors are launched.

It is obvious from that description that any interrupted net connection or other disturbances like system shutdowns etc. stops the netUpdate applications from reaching the next breakpoint. After a restart of the netUpdate application and the necessary primary steps the netUpdate application proceeds from the last successful reached breakpoint.

### 3. Security

In such an open system defined here security –mostly attacks by viruses, worms Trojan horses or man-in-the-middle attacks- is of primary concern. It therefore have been built in several safe guards on various levels to ensure a protection towards potential attacks:

- before downloading any netUpdate update files, the validity of the requested URLs (`product:url` attribute) of the update files is checked against the registered URLs of these companies dynamically on the TMR site. This ensures protection against maliciously changed definition files by e.g. a virus installed on a local system. If the check of a update file URL in the TMR security database is not found or is invalid, then the project is thereafter not considered in the update space analysis. It's as if the project is faded out from the available list of projects seen from the `netUpdate_master`. Of course, all dependencies and post processor directives are ignored as well under this condition.
- running an interprocess communication with a proprietary handshaking protocol prior to any netUpdate tasks checks the authentication of `netUpdate_clients`.

Due to these security measurements the authentication of the in and outbound communication can be enforced as much as possible. Nevertheless, the authentication of the `netUpdate_master` or any malicious actions scripted in the update file of a registered company can not be guaranteed. Of course, there can be never an absolute guarantee for any scripting system doing outbound communication at least from a principal point of view. It is a fact of life to coexist with security threads with the presence of open TCP/IP communication systems (as most already learned by using any email system) – therefore the real lesson learned is ...



## 4. Reference documentation of the schema definitions

### 4.1 Conventions, terms, and definitions used

This paragraph describes the conventions, terms, and definitions used in the following paragraphs for documenting the XSD schema files involved in the netUpdate project. It is assumed that the reader has at least a basic understanding of XSD schemas. Therefore, if there is an ambiguous description or insufficient documentation regarding the XSD schemas itself, you need to look up the official XSD schema definitions at W3C (<http://www.w3c.org/>).

#### Conventions

- the used character set in all schema files or accompanying data file (XML files) is **Unicode UTF-8**
- the description of the elements is in alphabetical order
- [min,max] depicts the minimal and maximal number of occurrences of this element. Note that min and max are  $\in \mathbf{N}$  and  $\min \leq \max$ .
- content of element: depicts the type of element. 4 different types can be distinguished, namely
  - textonly: the content consists of text
  - elementOnly: the content is defined in terms of other elements
  - mixed: the content is built by text and other elements
  - empty: there is neither text nor elements in that element, that is the element consists only of attributes
- default of attributes: gives an implicitly set default for the attribute

#### Data types used

name	description
boolean	boolean has the value space required to support the mathematical concept of binary-valued logic: {true, false}. An instance of a data type that is defined as boolean can have the following legal literal value {true, false, 1, 0}.
double	The double data type corresponds to IEEE double-precision 64-bit floating point type [IEEE 754-1985]. The basic value space of double consists of the values $m \times 2^e$ , where m is an integer whose absolute value is less than $2^{53}$ , and e is an integer between -1075 and 970, inclusive.
enumeration	enumeration constrains the value space to a specified set of values.
long	long is derived from integer by setting the value of maxInclusive and minInclusive to be 9223372036854775807 and -9223372036854775808.
NMTOKEN	NMTOKEN represents a token concept. The value space of NMTOKEN is the set of tokens (strings) that match the production.
pattern	pattern is a constraint on the value space of a data type, which is achieved by constraining the lexical space to literals, which match a specific pattern. The value of pattern must be a regular expression.
string	The string data type represents character strings. The value space of string is the set of finite-length sequences of characters. A character is an atomic unit of communication.
dateTime	dateTime represents a specific instant of time. The value space of dateTime is the space of combinations of date and time of day values as defined in [ISO 8601]. For example, to indicate 13:20 on May the 31st, 2001 for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 2001-05-31T13:20:00-05:00.
anyURI	anyURI represents a Uniform Resource Identifier Reference (URI). An anyURI value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be a URI Reference). This type should be used to specify the intention that the value fulfils the role of a URI as defined by [RFC 2396], and as amended by [RFC 2732].

**Caution!**

All local path names and parameters (even the fully qualified registry keys) are formed using an Unix-like notation with slashes (/) – the backslash is not allowed!

E.g. C:\test\data\file.dat *legal notation*  
 C:/test/data/file.dat *illegal notation*

**4.2 netUpdate\_def.xsd – netUpdate definition schema**

The root element of a netUpdate definition file is `definition`. All other elements are directly or indirectly referenced therein.

In the following section all elements of the schema definition are documented in alphabetical order. Please note that the normative definition is always the `netUpdate_def.xsd` schema file itself!

**Preamble**

The preamble of the schema file is defined as follows:

```
<schema
  targetNamespace="http://www.netupdate.ch/xsd/netupdate"
  xmlns:netupdate="http://www.netupdate.ch/xsd/netupdate"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
```

From this definition it is evident that all elements must be qualified that is defined with the namespace tag followed by the element name. Furthermore, the target namespace and the specific name space `netupdate` maps to `http://www.tarmed.net/xsd/netupdate`. Therefore, to define a correct data file the XML elements must be defined inside the following XML code snippet:

```
<?xml version="1.0" standalone="no"?>
<netupdate:definition
  xmlns:netupdate="http://www.netupdate.ch/xsd/netupdate"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.netupdate.ch/xsd/netupdate
  netupdate_def.xsd">
```

*XML elements go here...*

```
</netupdate:definition>
```

**Element company**

name	data type	content of element
company	-	empty

`company` carries some information about the company developing the product. This information will be displayed upon request in the netUpdate application.

**subelements in the requested order**

none

attribute name	data type	[min,max]	attribute default
email	pattern	[0,1]	-
email=(.+@.+ ) is the primary contact email of the company.			
name	string	[1,1]	-
name gives the name of the company.			
phone	string	[0,1]	-
phone is the primary contact phone number of the company.			
url	anyURI	[0,1]	-
url is an absolute URL to the website of that company.			

**Element definition**

name	data type	content of element
definition	-	elementOnly

definition is the root block of this schema and consists of the parameter, product and company information, respectively.

**subelements in the requested order**

```

parameter [0,n]
product [1,1]
company [1,1]
    
```

attribute name	data type	[min,max]	attribute default
none			

**Element parameter**

name	data type	content of element
parameter	-	empty

parameter defines one symbol to value assignment. The value can be built from prior defined symbols, literals, or full registry keys.

**subelements in the requested order**

none

attribute name	data type	[min,max]	attribute default
symbol	pattern	[1,1]	-

symbol=`([A-Z][A-Z0-9_-]{0,})` is the name of the symbol. Legal symbol names begin with an uppercase character followed by any sequence of uppercase characters numbers or the underscore and dash character. These parameters are used as transport mechanism for license information in `product:url`.

attribute name	data type	[min,max]	attribute default
value	pattern	[1,1]	-

value=`([A-Za-z0-9_-/${}: ]+)` is the expansion of the symbol. value is built from prior defined symbols, literals, and full registry keys. Symbols and registry keys are supplied in the form of `#{PARAMETER_NAME}` or `#{REG_KEYNAME}` and can be nested as well. Please not that an Unix-like notation with slashes (/) is used for building path names – the backslash is not allowed!

**Element product**

name	data type	content of element
product	-	empty

product defines the product in terms of the name, error communication, and URL for accessing the update information (cf. the netUpdate\_upd.xsd schema file).

**subelements in the requested order**

none

attribute name	data type	[min,max]	attribute default
ean	pattern	[1,1]	-

ean=`(76[0-9]{11})` is the official EAN number of the company (global identification number in official EAN lingua) being responsible for the product. This EAN number and the base URL is checked dynamically on the TMR site against a registration database prior to any update process.

attribute name	data type	[min,max]	attribute default
email	pattern	[1,1]	-

email=`(.+@.+)` is the email address to which an error report is automatically sent if and only if an error occurs during the update process that is related to the data or web services for that product. That is if an application error occurs, then this service is not activated. The primary motivation for this is to keep you as the owner of the software informed about possible problems.

attribute name	data type	[min,max]	attribute default
name	string	[1,1]	-

`name` supplies the name of the product. This name is displayed as entry in the netUpdate application. While not mandatory it is recommended to choose this name as part of the definition file name as well (e.g. `name_dt.def` for the german version of this product)

<code>url</code>	<code>anyURI</code>	<code>[1,1]</code>	-
------------------	---------------------	--------------------	---

`url` is the absolute URL where to retrieve the update file. Note that one can submit local information like license keys by defining a appropriate parameter and the using a http GET syntax for transporting that parameter to the netUpdate\_updServer. Before sending out the request, the netUpdate application will expand the parameter to the correct value to help authenticate the user on the target update server. For instance such an URL can be written as:

`http://www.test.net/test/test.asp?UID=${UID}`

Please note that if a URL has more the just 1 parameter then the & between any to arguments –due the schema definition- must be escaped by the entity `&amp;`!

### 4.3 netUpdate\_upd.xsd – netUpdate update schema

The root element of a netUpdate update file is `update`. All other elements are directly or indirectly referenced therein.

In the following section all elements of the schema definition are documented in alphabetical order. Please note that the normative definition is always the netUpdate\_upd.xsd schema file itself!

#### Preamble

The preamble of the schema file is defined as follows:

```
<schema
  targetNamespace="http://www.netupdate.ch/xsd/netupdate"
  xmlns:netupdate="http://www.netupdate.ch/xsd/netupdate"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
```

From this definition it is evident that all elements must be qualified that is defined with the namespace tag followed by the element name. Furthermore, the target namespace and the specific name space `netupdate` maps to `http://www.tarmed.net/xsd/netupdate`. Therefore, to define a correct data file the XML elements must be defined inside the following XML code snippet:

```
<?xml version="1.0" standalone="no"?>
<netupdate:update
  xmlns:netupdate=" http://www.netupdate.ch/xsd/netupdate "
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.netupdate.ch/xsd/netupdate
  netupdate_upd.xsd">
```

*XML elements go here...*

```
</netupdate:update>
```

#### Element collection

name	data type	content of element
<code>collection</code>	-	<code>elementOnly</code>

`collection` is a container element for the purpose of incremental updates e.g. of a database. Note that each defined collection defines a fallback solution (`fallback`) that is used if none of the increments (`increment`) can be used. It is therefore essential that the fallback solution supplies all data e.g. the full database in the case of an incremental database update. If at least 1 increment is defined, then a handler for processing the increments must be given as well.

#### subelements in the requested order

```
(increment [1,n], handler [1,1]) [0,1]
  fallback [1,1]
```

attribute name	data type	[min,max]	attribute default
<code>increment_basedate</code>	<code>dateTime</code>	<code>[0,1]</code>	-

`increment_basedate` explicitly defines the lower date boundary of the 1<sup>st</sup> increment. If this date is not given, the the 1<sup>st</sup> increment spans a date range of zero!

### Element definition

name	data type	content of element
definition	-	empty

`definition` is the element that defines the netUpdate definition file of the focused product in terms of a special resource element.

#### subelements in the requested order

none

attribute name	data type	[min,max]	attribute default
last_modified	long>0	[1,1]	-

`last_modified` is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).

attribute name	data type	[min,max]	attribute default
local_name	pattern	[1,1]	-

`local_name=(^${NETUPDATE_DATA}[A-Za-z0-9_-]+\.\def)` is the local absolute file name of the definition file located in the NETUPDATE\_DATA directory. Note that the file extension must be `.def!`

attribute name	data type	[min,max]	attribute default
remote_name	anyURI	[1,1]	-

`remote_name` is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!

attribute name	data type	[min,max]	attribute default
size	long>0	[1,1]	-

`size` is the size of the resource in bytes.

### Element fallback

name	data type	content of element
fallback	-	empty

`fallback` defines all needed information of a fallback resource such that a decision for a smart update is possible. Note that if no increment element is available or if `none` can be used for the upgrading then the fallback element is taken. Therefore the fallback should always be the full up-to-date resource, not a partial increment! If the fallback is applicable then the handler application is not launched as the fallback is *per definition* not an increment!

#### subelements in the requested order

none

attribute name	data type	[min,max]	attribute default
flag	NMTOKEN	[0,1]	mightUpdate



flag=[mightUpdate| mightInstallWithPathGeneration| mightInstallIfPathExist| mustInstallWithPathGeneration| mustInstallIfPathExist| mustUpdateWithPathGeneration |mustUpdateIfPathExist] gives the applicable type of update.

The meaning of flag is as follows:

- an unconditional install is performed if the installation flag is set to “mustInstallWithPathGeneration” if and only if the resource does not exist locally. The directory path will be generated if it does not already exist,
- an unconditional install is performed if the installation flag is set to “mustInstallIfPathExist” if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install,
- an unconditional update is performed if the installation flag is set to “mustUpdateWithPathGeneration” even if the resource or the directory path does not exist (the path will be generated if it does not already exist),
- an unconditional update is performed if the installation flag is set to “mustUpdateIfPathExist” even if the resource does not exist locally. However, the directory path must already exist to perform the forced update,
- an unconditional install is performed if the installation flag is set to “mightInstallWithPathGeneration” if and only if the resource does not exist locally. The directory path will be generated if it does not already exist. In the case that the resource does exist, then a conditional update is performed under the same conditions as with “mightUpdate”,
- an unconditional install is performed if the installation flag is set to “mightInstallIfPathExist” if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install. In the case that the resource does exist, then a conditional update is performed under the same conditions as with “mightUpdate”,
- a conditional update is performed if the installation flag is set to “mightUpdate” (the default value) if and only if the corresponding resource exists locally and the last modified date of that local resource is less than the remote modification date.

last_modified	long>0	[1,1]	-
---------------	--------	-------	---

last\_modified is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).

local_name	pattern	[1,1]	-
------------	---------	-------	---

local\_name=(`[^\|]+`) is an absolute local file path of the resource. The attribute can include any numbers of previously defined parameters declared in the parameter section as parameter expansion is carried out. Parameters are supplied as `#{PARAMETER_NAME}`.

remote_name	anyURI	[1,1]	-
-------------	--------	-------	---

remote\_name is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!

size	long>0	[1,1]	-
------	--------	-------	---

size is the size of the resource in bytes.

### Element handler

name	data type	content of element
handler	-	empty

handler defines all needed information of an executable that is able to incrementally update a collection resource by using the given increments. The handler is like a post\_processor element but tightly coupled to a collection.

#### subelements in the requested order

none

attribute name	data type	[min,max]	attribute default
last_modified	long>0	[1,1]	-

last\_modified is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).

local_name	pattern	[1,1]	-
------------	---------	-------	---

`local_name=(^\\+)` is an absolute local file path of the target resource holding all information (e.g. the database file). The attribute can include any numbers of previously defined parameters declared in the `parameter` section as parameter expansion is carried out. Parameters are supplied as `${PARAMETER_NAME}`.

<code>remote_name</code>	anyURI	[1,1]	-
<code>remote_name</code> is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!			
<code>size</code>	long>0	[1,1]	-
<code>size</code> is the size of the resource in bytes.			

**Element increment**

name	data type	content of element
increment	-	empty

`increment` defines all needed information of one increment resource such that a decision for a smart update is possible. By defining an increment information a handler resource (an executable) must be defined as well that is able a incrementally update the target using the increment resource!

**subelements in the requested order**  
none

attribute name	data type	[min,max]	attribute default
<code>last_modified</code>	long>0	[1,1]	-
<code>last_modified</code> is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).			
<code>local_name</code>	pattern	[1,1]	-
<code>local_name=(^\\+)</code> is an absolute local file path of the target resource holding all information (e.g. the database file). The attribute can include any numbers of previously defined parameters declared in the <code>parameter</code> section as parameter expansion is carried out. Parameters are supplied as <code>\${PARAMETER_NAME}</code> .			
<code>remote_name</code>	anyURI	[1,1]	-
<code>remote_name</code> is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!			
<code>size</code>	long>0	[1,1]	-
<code>size</code> is the size of the resource in bytes.			

**Element parameter**

name	data type	content of element
parameter	-	empty

`parameter` defines one symbol to value assignment. The value can be built from prior defined symbols, literals, or full registry keys.

**subelements in the requested order**  
none

attribute name	data type	[min,max]	attribute default
<code>symbol</code>	pattern	[1,1]	-
<code>symbol=([A-Z][A-Z0-9_-]{0,})</code> is the name of the symbol. Legal symbol names begin with an uppercase character followed by any sequence of uppercase characters numbers or the underscore and dash character. These parameters are used in the definition of local file names (e.g. <code>resource:local_name</code> ).			
<code>value</code>	pattern	[1,1]	-

value=( [A-Za-z0-9\_-/\${}: ]+) is the expansion of the symbol. value is built from prior defined symbols, literals, and full registry keys. Symbols and registry keys are supplied in the form of \${PARAMETER\_NAME} or \${REG\_KEYNAME} and can be nested as well. Please not that an Unix-like notation with slashes (/) is used for building path names – the backslash is not allowed!

**Element post\_processor**

name	data type	content of element
post_processor	-	empty

post\_processor defines an application that is started after the update of the „target“ project is carried out. If no resource of that project is updated then the post\_processor is not launched!

**subelements in the requested order**

none

attribute name	data type	[min,max]	attribute default
flag	NMTOKEN	[0,1]	mightUpdate

flag=[mightUpdate| mightInstallWithPathGeneration| mightInstallIfPathExist| mustInstallWithPathGeneration| mustInstallIfPathExist| mustUpdateWithPathGeneration |mustUpdateIfPathExist] gives the applicable type of update.

The meaning of flag is as follows:

- an unconditional install is performed if the installation flag is set to “mustInstallWithPathGeneration” if and only if the resource does not exist locally. The directory path will be generated if it does not already exist,
- an unconditional install is performed if the installation flag is set to “mustInstallIfPathExist” if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install,
- an unconditional update is performed if the installation flag is set to “mustUpdateWithPathGeneration” even if the resource or the directory path does not exist (the path will be generated if it does not already exist),
- an unconditional update is performed if the installation flag is set to “mustUpdateIfPathExist” even if the resource does not exist locally. However, the directory path must already exist to perform the forced update,
- an unconditional install is performed if the installation flag is set to “mightInstallWithPathGeneration” if and only if the resource does not exist locally. The directory path will be generated if it does not already exist. In the case that the resource does exist, then a conditional update is performed under the same conditions as with “mightUpdate”,
- an unconditional install is performed if the installation flag is set to “mightInstallIfPathExist” if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install. In the case that the resource does exist, then a conditional update is performed under the same conditions as with “mightUpdate”,
- a conditional update is performed if the installation flag is set to “mightUpdate” (the default value) if and only if the corresponding resource exists locally and the last modified date of that local resource is less than the remote modification date.

last_modified	long>0	[1,1]	-
---------------	--------	-------	---

last\_modified is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).

local_name	pattern	[1,1]	-
------------	---------	-------	---

local\_name=( [^\]+) is an absolute local file path of the resource. The attribute can include any numbers of previously defined parameters declared in the parameter section as parameter expansion is carried out. Parameters are supplied as \${PARAMETER\_NAME}.

remote_name	anyURI	[1,1]	-
-------------	--------	-------	---

remote\_name is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!

size	long>0	[1,1]	-
------	--------	-------	---

size is the size of the resource in bytes.

**Element resource**

name	data type	content of element
resource	-	empty

resource defines all needed information of one resource such that a decision for a smart update is possible.

**subelements in the requested order**

none

attribute name	data type	[min,max]	attribute default
flag	NMTOKEN	[0,1]	mightUpdate

flag=[mightUpdate| mightInstallWithPathGeneration| mightInstallIfPathExist| mustInstallWithPathGeneration| mustInstallIfPathExist| mustUpdateWithPathGeneration |mustUpdateIfPathExist] gives the applicable type of update.

The meaning of flag is as follows:

- an unconditional install is performed if the installation flag is set to "mustInstallWithPathGeneration" if and only if the resource does not exist locally. The directory path will be generated if it does not already exist,
- an unconditional install is performed if the installation flag is set to "mustInstallIfPathExist" if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install,
- an unconditional update is performed if the installation flag is set to "mustUpdateWithPathGeneration" even if the resource or the directory path does not exist (the path will be generated if it does not already exist),
- an unconditional update is performed if the installation flag is set to "mustUpdateIfPathExist" even if the resource does not exist locally. However, the directory path must already exist to perform the forced update,
- an unconditional install is performed if the installation flag is set to "mightInstallWithPathGeneration" if and only if the resource does not exist locally. The directory path will be generated if it does not already exist. In the case that the resource does exist, then a conditional update is performed under the same conditions as with "mightUpdate",
- an unconditional install is performed if the installation flag is set to "mightInstallIfPathExist" if and only if the resource does not exist locally. However, the directory path must already exist to perform the forced install. In the case that the resource does exist, then a conditional update is performed under the same conditions as with "mightUpdate",
- a conditional update is performed if the installation flag is set to "mightUpdate" (the default value) if and only if the corresponding resource exists locally and the last modified date of that local resource is less than the remote modification date.

last_modified	long>0	[1,1]	-
---------------	--------	-------	---

last\_modified is the last modified date of the resource given as an epoch time (number of seconds since 1.1.70).

local_name	pattern	[1,1]	-
------------	---------	-------	---

local\_name=(<sup>[^\\]</sup>+) is an absolute local file path of the resource. The attribute can include any numbers of previously defined parameters declared in the parameter section as parameter expansion is carried out. Parameters are supplied as \${PARAMETER\_NAME}.

remote_name	anyURI	[1,1]	-
-------------	--------	-------	---

remote\_name is an absolute URL giving the location where to retrieve the resource. Note that this attribute must be a legal URL, parameters are not allowed here!

size	long>0	[1,1]	-
------	--------	-------	---

size is the size of the resource in bytes.

**Element update**

name	data type	content of element
update	-	elementOnly

update is the root block that defines all bits and pieces for the update of that product.

**subelements in the requested order**

```

        parameter [0,n]
        definition [1,1]
            uses [0,n]
    (resource|collection) [1,n]
    
```

attribute name	data type	[min,max]	attribute default
valid	DateTime	[0,1]	-

valid is a timestamp defining the start of validity of the update of that product. That is, when given any update process prior to the given time is not performed – there will be no update check at all!

**Element uses**

name	data type	content of element
uses	-	elementOnly

uses defines a parent-child relationship or dependency. Such a relationship means that the product uses and relies upon another product. Implicitly you express with such a relationship that the other product must be checked for possible updates as well (if it does exist).

**subelements in the requested order**

```

        post_processor [0,n]
    
```

attribute name	data type	[min,max]	attribute default
definition	pattern	[1,1]	-

definition=(^\${NETUPDATE\_DATA}[A-Za-z0-9\_-]+\.\def) is the definition file of the other product the current product is relying on. Note that the definition file must be located in the NETUPDATE\_DATA directory and that the file extension must be .def!

post_processor	string	[0,1]	-
----------------	--------	-------	---

post\_processor is the full name (path and filename) of an application that is started after the update of the „child“ product is carried out. If no resource of that product is updated then the post\_processor is not launched. ! The attribute can include any numbers of previously defined parameters declared in the parameter section as parameter expansion is carried out. Parameters are supplied as \${PARAMETER\_NAME}.

## 5. Version history

This paragraph gives a history about the changes of the netUpdate project as a function of the version. The history is subdivided into a table for each schema file and a remaining table for changes apart from schema modifications.

### Changes apart from schema modifications

Date	Version	Content/change
10.04.04	1.00	initial version

### netUpdate\_def.xsd

Date	Version	Content/change
10.04.04	1.00	initial version

### netUpdate\_upd.xsd

Date	Version	Content/change
10.04.04	1.00	initial version