Enterprise Server Synchronization and Event Publishing Deployment Guide

Publication Information

Release: Enterprise Server 10.5.1.1

Publication date: April 11, 2017

Document code: U8780B US

Contents

About this guide

This document explains how to use the synchronization methods and the underlying Synchronization Server for Enterprise Server 7 SP 1. The Synchronization Server is a part of the Data Director (da) package. In Enterprise Server 7, the 3.3 version of this package is included.

This document contains an overview of event publishing and the Synchronization Server. Also is explained what happens within LN. This includes details on how to set up the LN environment for event publishing, how to deploy the Synchronization Server at runtime, and guidelines on how to deal with problems.

For a detailed description of the available Synchronization Server sessions see the online help.

Related to application development, this document provides information on developing publishing methods for a business object.

Intended audience

This guide is intended for system administrators.

Related documents

You can find the documents in the product documentation section of the Infor Xtreme Support portal, as described in "Contacting Infor" on page 9.

Contacting Infor

If you have questions about Infor products, go to the Infor Xtreme Support portal at www.infor.com/inforxtreme.

If we update this document after the product release, we will post the new version on this Web site. We recommend that you check this Web site periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

Chapter 1 Introduction

1

Business data and methods as available in LN are grouped in business objects. A business object (also known as BDE, business data entity) is capable of publishing events when changes occur to its instances. A client application can request changes on such a business object to be published. In that case, when a new instance of a business object is created or an existing instance of a business object is changed or deleted then a create, change or delete event is published. The Synchronization Server enables publishing of these events from LN.

Definitions, Acronyms and Abbreviations

This table shows the abbreviations that are used in this document:

Term	Definition
BDE	Business Data Entity, also known as business object
BOL	Business Object Layer
BOR	Business Object Repository
LN	The LN enterprise resource planning product
VRC	Version - Release - Customer, which together define the edition of a software component in LN
XSD	XML Schema Definition

Chapter 2 Overview

2

Business objects offer methods to retrieve information or perform a specific action for a business object. For publishing, the available methods are *PublishList*, *PublishChanges* and *UnpublishChanges*.

The client application sends the *PublishList* or *PublishChanges* request to LN and is listening to the communication channel (bus component) that is used to transfer the resulting data.

Upon *PublishList*, a process is started that collects the data in accordance with selection and filter. The business objects are published by publishing messages containing the data to the listener. In case of a large data set, multiple messages can be sent. The client applies the data to its own database and removes instances that are missing (so they are not relevant anymore).

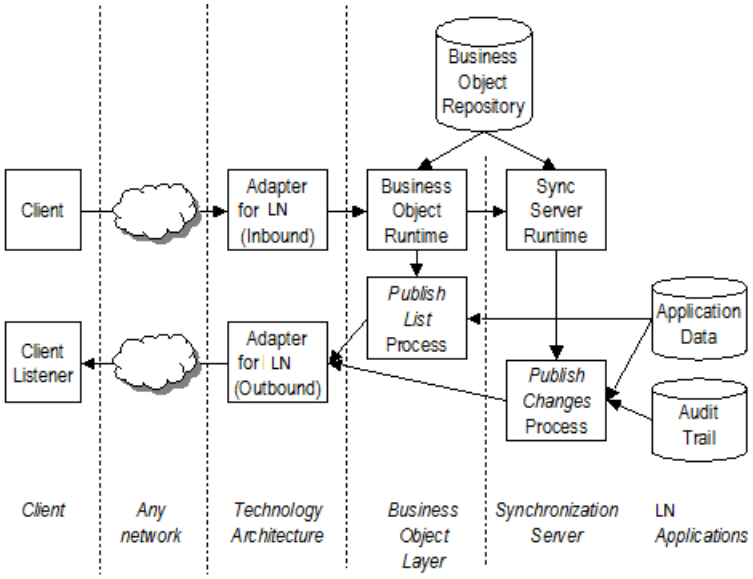
This process stops as soon as it completed processing the whole data set in accordance with the specified selection and filter.

The selection specifies what parts (attributes) of a business object must be published. For example, the order number, order date and status from an order and the line number, item, quantity and price for each order line. The filter specifies what instances of the object and its components must be included. For example, only include orders having status ‘open’ and only include order lines having a quantity more than 10.

Upon *PublishChanges*, a process is started that detects any new events (either create, change or delete events). Events that meet the specified selection and filter are published.
The process is stopped by invoking the *UnpublishChanges* method. The settings are changed by invoking the *PublishChanges* method using a different selection and/or filter.

The client receives the data set or change events and takes action upon these. For example, by synchronizing its own database or by starting an application process based on an incoming event.

This diagram provides an overview of the components involved:



The client and the LN backend communicate using a middleware infrastructure consisting of the network and additional technology such as a brokering solution. This infrastructure is enabled by the SSA Technology Architecture. This allows the client to send the *PublishList* and *PublishChanges* requests and to receive the resulting data or change events.

The *PublishList* is implemented in LN by a process that is started in the Business Object Layer.

For the *PublishChanges* in LN, the implementation is delegated to the Synchronization Server, which starts a process to detect events in the application data. The change events are detected through auditing. While transactions are executed by user sessions or other application processes, relevant changes are written to an audit trail. This is the only overhead on the end user. An offline process picks up the transaction data that is needed for the business object and in accordance with the specified selection and filter. An event message is created representing the event and its data and that message is published.

The *PublishList* automatically stops if the requested data set is processed. Regarding *PublishChanges*, the publishing of events is a continuous process, because new change events may always occur. It is stopped when the client sends an *UnpublishChanges* request.

Synchronization Server

The Synchronization Server implements the publishing of event messages for LN. This is done by creating a synchronization object for a business object. A synchronization object is the selection of a business object, extended with selection of components and attributes and (optionally) a filter. The synchronization object is enables the synchronization at runtime for the selected business object.

If LN must be synchronized with another application, the data source at the LN side is actually a set of table fields. However, the LN tables are not directly accessible. LN tables can be accessed

through business objects. To read the fields in a table, you must call a method of a business object. This method handles the attributes of the business objects. The attributes of the business object are mapped to table fields, but formatting or calculation may be involved.

The Synchronization Server takes care of:

- Event detection

  Changes in the underlying backend components of the business objects are detected. If a change is applied to a certain column in a table and that column is mapped to an attribute of a business object, from which a client wishes to see changes pushed, the Synchronization Server detects that the change has happened. In LN this is done using audit functionality.

- *Event filtering*

  Create, change or delete events are filtered based on values of attributes, such as is possible in the *List* and *Show*. Create and delete events are either passed on or skipped. But in case of change events it is additionally possible that the event is passed on but altered. If the original values of the attributes (before the change) meet the filter, but the current values (after the change) do not, the event is passed on as a delete event instead of a change event. Similarly, if the original values of the attributes (before the change) do not meet the filter, but the current values (after the change) do, the event is passed on as a create event.

- *Event aggregation*

  A change event at database level (table column) is converted into a change event for a business object (component attribute). By default, changes in attributes are aggregated towards pushing out all relevant surrounded component data. So even if one attribute is changed it will push out the contents of the other attributes inside the same component as well. In addition to adding attributes for the changed component(s), the aggregation process also adds parent components with at its attributes.
  This default behaviour can be changed, by specifying at configuration time that only certain attributes must be pushed instead of all data of the entire component involved, or by specifying that also data from other unchanged child components must be included.
  For example: A change of a quantity in a sales order lines table, are transformed into an XML document containing the sales order object consisting of the (unchanged) header and the changed order line.

- *Event publishing*

  The XML document that describes the change event is pushed out to an external component that is responsible for handling the event in a correct way. This component is responsible for feeding the data further into a broker product, or saving it in a file on the file system, or simply feeding it into a message queue, for example.

  The destination is specified using a so-called *bus component*, which actually refers to the listening party that must receive the event messages.

Usage of Event Publishing

Event publishing has these types of usage:

- Event handling: use events from LN to take action. For example, if a new quotation is created in LN initiate an approval process.
- Synchronization: use events from LN to synchronize business data in another application or database.

Event Handling

Event handling is required when an application needs to be notified of a specific event in LN. For example, the creation of a new sales order, the completion of a production order, or the blocking of a customer.

In that case the *PublishChanges* can be used. Events on a specific business object are published and can be processed by, for example, a workflow application.

Alternative options for implementing events are:

- DBMS triggers. This is a very direct way to react to a database change, and customer-specific code can be included.

  Disadvantages of this approach are:

  - Difficult to implement and maintain, triggers are defined at a technical level (even more technical than the LN database model)
  - Consistency issues: the trigger is executed before commit, so must rollback the event if the transaction fails
  - Overhead on end-user processes, because the event is created in the user transaction.
- Send events from an application by customizing the application. This is more high-level. In some cases, this approach is useful, especially if the event is linked to a specific action in one specific session or program.

  Disadvantages of this approach are:

  - Customization of application reduces maintainability and increases total cost of ownership
  - Development (design and coding) is needed.
  - Knowledge is required to find right place to implement this; in legacy applications it is often difficult to implement because the conditions that should trigger an event may be implemented in multiple places in multiple programs.

Synchronization

Synchronization is a type of integration where the involved sites or applications have an overlap in persistent data. For example, in addition to an ERP application containing item (product) data, a customer relationship management application is used that also contains product information.

Basically, two approaches exist for this type of integration:

- Regularly resending the complete data set containing all relevant instances of the business objects.
- In change-based synchronization, instead of resending the whole data set regularly, the target applications receive change events, such as 'create', 'change' or 'delete' for those object instances that were changed. In this case a one-time synchronization of all relevant instances of the business objects is done in order to get a baseline, followed by synchronization of relevant changes whenever they occur.

  The first approach can be implemented by regularly issuing a *PublishList* request, which sends the requested data set. The second approach can be implemented by using a combination of *PublishList* and *PublishChanges*.

Alternative options for implementing synchronization are:

- Use the *List* method. This is comparable to using *PublishList*, except that the data is immediately returned in the response. This is easier from a setup perspective, because no publishing setup is required (i.e. defining bus component). In case of large data sets, the advantage of publishing is that the requesting application doesn’t have to wait for the response to arrive and doesn’t have to issue multiple requests in case the data set is sent in multiple chunks.
- At a lower (more technical) level, the LN Exchange module can be used to export and import LN data. This is configured based on database tables instead of business objects, so more technical knowledge is needed. Additionally, it is not using XML and consequently is less open. It is currently not fully incorporated in the BDE standards and the corresponding interoperability architecture. Advantages of Exchange are that is can handle ‘any’ classical file format (fixed length or using a separator), and it is fast.
- At an even lower level, mirroring or replication at DBMS level can be used. This is only suitable for homogenous environments. For example, if the database at one site is an exact copy of the database at another site.
- Use the *List* method in combination with DBMS triggers or an application customization to receive the events.

Synchronization versus Tight Application Integration

Synchronization as described earlier is usually needed because applications are developed separately, using their own database schema. However, if applications are designed jointly, synchronization may not be needed.

For example, if a product lifecycle management application and a sales application are designed, the first one will contain objects such as product (item), engineering data, bill of materials, while the second one will contain customers, quotations and sales orders. If both applications are in a local area network, the sales application does not need duplicate data from the product lifecycle

management application. Instead it refers to data from that application. For example, a sales order refers to one or more products.

In the same way, when using another application as an add-on to LN, it may not always be required to use a persistent database for data that is already available in LN, but the ask for data (or updates) when needed.

In this case, the *PublishList* or *PublishChanges* are not used for synchronization. Instead, methods such as *List* and *Show* (and probably also *Create*, *Change* and *Delete*) are used, because the applications have a tight integration at runtime. Note however that event handling as described in “Event Handling” can still be required.

This table shows a number of differences between data synchronization on the one hand and other methods to retrieve data on the other.

Topic	List, Show etc.	Synchronization methods
Goal	Meant for applications that do not locally store data that is in LN as well, but only keep references to LN objects. Those applications browse through LN objects and get data details only when required.	Meant for synchronization of LN and other applications. Can also be used to notify other applications of certain situations in the LN application's business objects.
Frequency	Used ad hoc.	Used regularly.
Configuration	Generic methods; requirements (what attributes are required; what filters must be applied etc.) are partly determined at the moment a client application actually sends the question message. Subsequent calls for the same business object may have different requirements regarding the attributes that are required or the filters that must be applied.	Configuration takes place at implementation time. The required attributes and filters are defined once and are more or less static.
Consistency	No special measures are taken to guarantee consistency. Only the internal consistency of the data set for a single business object is taken care of.	Consistency must be guaranteed amongst related business objects. For example, if a client application requires item and production order data, it should not receive production orders if the referred items are not yet sent to the client. Also the client must keep track of its status to ensure no data or no changes are missing.

Topic	List, Show etc.	Synchronization methods
Data set size	The number of object instances retrieved is limited.	The number of object instances can be very high, especially when synchronizing dynamic data. It must be possible to retrieve large data sets in chunks.
Optimization	Optimization is limited, because you do not know when a method is called and what will exactly be asked. (Although in special cases customized methods can be implemented.)	Measures are taken to decrease latency and reduce network and system load. For example, by sending changes (deltas) instead of the complete set of objects.

Chapter 3 Publishing Interface – How to Use

3

How to use the publishing interface, from a design perspective and from the client application's point-of-view. Additionally, a number of important things to take into account when using the publishing methods are discussed. To check the status of the LN server after issuing a *PublishList* or *PublishChanges* request, is also described.

Designing an Integration

When designing the integration:

1. Determine the required business objects for the integration. This procedure assumes that the required business objects and their components and attributes are available in the Business Object Repository. If they are not, they must be defined there. This is out of scope for this document.
2. For each business object, determine:
   - What attributes are required. For example, for the Item business object only include attributes that are relevant for sales.
   - What filters must be applied. For example, for the sales order business object only include 'open' sales orders.
3. Group the business objects into related business objects that must be synchronized together. For example, items and sales orders will probably have to be in one group. If the items are synchronized once a week and the sales orders every day, you may end up with sales orders that refer to an item that is not yet synchronized.
4. Estimate the optimal synchronization frequency. What is the allowed latency for the synchronization objects involved? Is it sufficient to synchronize the applications once per month, once per week, once per day, or once per hour?

   When choosing a synchronization frequency, also take the costs into account. For example, do the benefits of a more frequent synchronization outweigh the extra costs, network load and system load? Additionally, how should the load for the network be distributed? Do you must synchronize at predefined quiet moments or do you prefer an approach where the load is spread over time as much as possible?

5 For each synchronization object, choose the appropriate synchronization method. Synchronization can either be done by sending the complete set of data or by sending the changes on the data. In short, these rules of thumb apply:

- If a lot of objects exist in the database and a relatively(!) small part of these objects is changed during one period (based on the synchronization frequency), an event-based approach (using the *PublishChanges* method) must be considered.
- If the data set is small or only few objects are unchanged during a period, a full synchronization (using the *List* or *PublishList* method) is likely to be the most appropriate method.

If only one client uses the *PublishChanges* for a specific business object, the settings can be tuned to the requirements of that client. However, if multiple clients (sites or applications) exist that must synchronize data using *PublishChanges* for the same business object, the requirements for all clients must be taken into account. By default, only a single event publisher is running for a business object.

Finally, note that synchronization is done per LN company, so a *PublishChanges* or *PublishList* must be executed per company.

Ownership of Data

In the design, ownership of data must be taken into account.

When synchronizing data from and/or to LN no automatic conflict resolution is available. If the same object instance is updated at two sites, synchronization may result in the changes from one site being overwritten by the changes from the other site. Consequently a clear ownership must be defined for the business objects.

Note that you can implement fragmentation, which can be done in two ways:

- Each site owns different *instances* for the same business object. For example, use multiple ranges of identifying attributes (e.g. item codes) are used. Or depending on the state of the business object: an order is created at one site, as soon as the status becomes ‘released’ the ownership is transferred to another site where the order is planned and executed, and as soon as the status becomes ‘completed’ the ownership goes back to the first site, where the financial conciliation and archiving of the order is done.
- Each site owns different *attributes* for the same business object. For example, one site is allowed to maintain the attributes regarding *estimated* resources for a service order, while another site is allowed to maintain only the attributes regarding *actually spent* resources.

The Client Side

Here is described how to use the data or events as published by *PublishList* or *PublishChanges* at the client side. Technical details are not included, because they depend on the client platform.

Synchronizing using *PublishList*

The *PublishList* provides the current status of the requested data set in LN. Let's assume the data set as provided by the *PublishList* method is used for synchronizing the client's database, and for each instance in LN there is one instance in the client's database and vice versa.

In that case:

1. For each object instance that is received:
   - If it does not exist in the client's database, then add it;
   - If it already exists in the client's database, then 'replace' the existing instance.

   This means:

   a. updating already existing components

   b. adding not yet existing components

   c. deleting components that are not available in the published object instance (so they were apparently deleted in LN).

   In some cases, it is possible to simply delete the existing instance and add the new one that was received.
2. Each object instance that is available in the client's database but not in the published data set must be deleted from the client's database (because it was apparently deleted in LN).
   Note that *PublishList* may publish the data set in multiple messages; deleting irrelevant objects should not be done until the last message has been processed.

   The process at the client's site is more complex if there is no one-to-one relation between the objects in LN and the client's objects.

Synchronizing Multiple Objects using *PublishList*

In case multiple objects (classes) are synchronized, the dependencies (references) between those objects must be taken into account. For example, let's assume a *PublishList* is issued for items (products) and production orders. Each production order line refers to one or more items.

In that case the correct sequence for applying the data to the client's database is:

1. Add or update items based on the messages published for the item business object.
2. Add or update production orders based on the messages published for the production order business object.
3. Delete production orders that are not relevant anymore.
4. Delete items that are not relevant anymore.

The risk of retrieving orders for unknown items can be further reduced by 'swapping sequence': first request the orders and then (after some time, as required) request the items. But when receiving the data at the client site first apply the items then apply the orders.

Synchronizing using *PublishChanges*

When using *PublishChanges*, the client doesn't receive object instances, but change events on object instances. Standard events are; create (a new instance was created), change (an existing instance was updated) or delete (an instance is removed). A change event may include the creation or deletion of components (such as order lines). In case of a change, only part of the object data may be included. By default, only the changed components and their parents are available (see "Selection and Filter"). For details on change events, see "Messages".

When synchronizing based on change events, the changes must be applied in the correct sequence, which is the sequence in which they occurred in LN. LN guarantees that the changes on a business object are published in the sequence in which they originally occurred.

Let's assume the data set as provided by the *PublishChanges* method is used for synchronizing the client's database, and for each instance in LN there is one instance in the client's database and vice versa.

In that case the logic for processing the received events is if the event is a:

- Delete event then find the corresponding instance in the database and delete it.
- Create event then create the corresponding instance in the database.
- Change event then process the components in the event message. Each component is either unchanged, or changed, or created or deleted.

  Creating components is done top-down: before creating a component its parent component(s) must be created Deleting components must be done bottom-up: before deleting a component it's aggregated child components must be deleted.

Synchronizing by Combining *PublishList* and *PublishChanges*

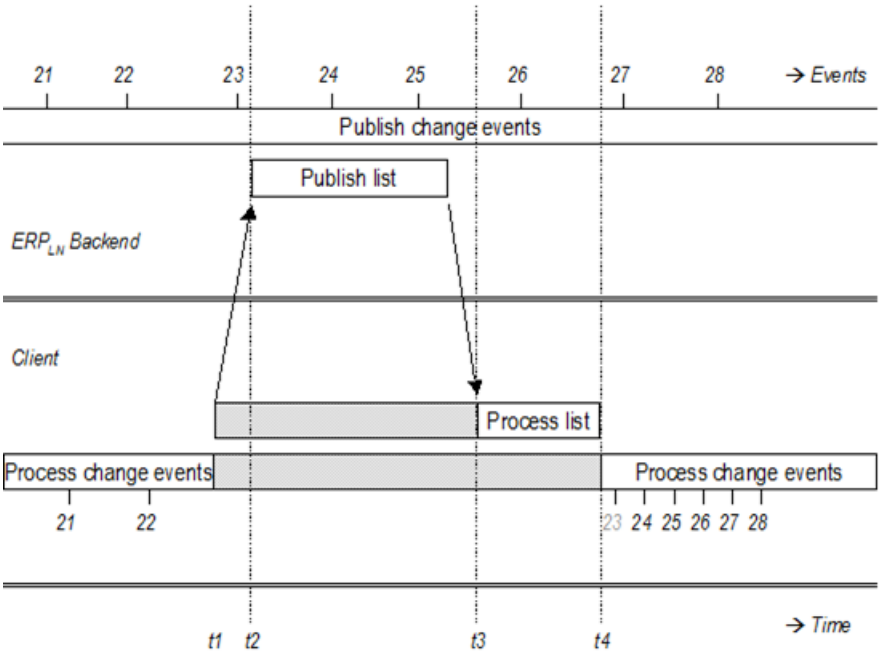
When synchronizing, the *PublishList* method can be used to retrieve the initial data set, while from that moment on the changes are processed based on the messages as provided using the *PublishChanges* method. Additionally, you can use *PublishList* also in a later stage for objects that are normally synchronized using *PublishChanges*. Normally this is not required, but if the client database got corrupted, or changes could not be applied properly, you can use *PublishList* to regenerate the complete data set.

When combining the *PublishList* and *PublishChanges* methods, the following must be taken into account.

The *PublishChanges* must be invoked before using *PublishList*. This is required because no change that is done after the *PublishList* process reads an object instance must be lost.

When issuing a *PublishList*, the client must stop processing the event messages from *PublishChanges*, until the complete data set from the *PublishList* was processed.

See this diagram:



The diagram assumes the event publishing is already running. An initial *PublishList* was done some time ago and the *PublishChanges* is running, at both the LN backend and the client.

At t1 the client temporarily stops processing the change events because a *PublishList* request is sent. Let's assume that events 21 and 22 were processed already, but event 23 is still on its way at that moment. This means events starting at event 23 must be buffered when received by the client, because LN will continue publishing client events.

At t2 (which is very close to t1) the *PublishList* process at the backend starts reading the data from the LN database. At t3 the client receives the data set from the *PublishList*. It applies the instances to its own database and at t4 this process is completed.

Then the processing of change events at the client is continued. The client will start processing the buffered change events.
Event 23 is skipped, because it has a timestamp that is before the timestamp that is included in the data set from the *PublishList*.
Event 24 and 25 will have to be processed, but the client must take into account that the instances changed by these events may have been read by the PublishList already after the moment the event occurred.
From event 26 onwards the event handling returned to normal.

Note that persistency is required to avoid losing events 23, 24, 25 and 26. If the middle tier (for example a message queuing system) does not provide this, the client must temporarily store incoming events that cannot yet be processed.

Synchronizing Multiple Objects using *PublishChanges*

When multiple business objects (classes) are synchronized using *PublishChanges*, the sequence must be taken into account if the business objects are related. For example, when synchronizing items and production orders, if in LN first an item is created and then a production order is generated for that item, the processing at the client site must be in the same sequence.

However, the LN backend does not guarantee the sequence of the messages for multiple business objects, even if they are published to the same bus component. Each business object will have its own publication process.

Consequently, the client must apply the change events in the correct sequence. This can be done using the eventTimestamp or eventTimestampSequenceNr as provided in the event message. The eventTimeStamp has a granularity of one second. The eventTimestampSequenceNr contains a unique sequence number for each transaction executed in LN, so it is more accurate to determine the sequence if multiple transactions were executed within a single second.

Note that if a client combines (net changes) multiple events on the same instance, the sequence is not valid anymore.

Event Handling

If the client does not use change events to synchronize its data, but to trigger a specific action, a different approach is used. The event as needed by the client must be defined in terms of standard create, change and delete events.

In some cases, this is simple. For example, if the client must take action when a new instance is created in LN. In some cases, it is more complex because also the state (attribute values) of the instance must be taken into account. For example, if the client must take action when a new sales order is entered manually or from a phone call, fax or mail. In that case the event type must be ‘create’ and the origin attribute must have value ‘Manual’, ‘Phone’, ‘Fax’ or ‘Mail’.

The requirements from the client must be included as much as possible in the selection and filter included in the *PublishChanges* request. However, this will not always succeed completely, due to the filtering limitations.

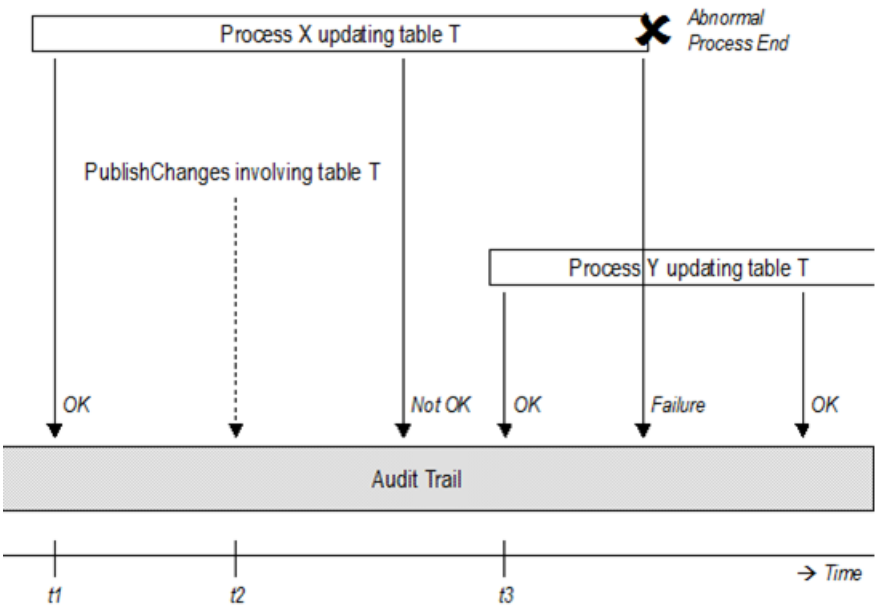
At runtime the client checks the incoming events and acts accordingly.

Important Things to Consider

Audit

Using PublishChanges on a live environment?

PublishChanges results in an audit profile being created and a create runtime for the audit profiles done. For that reason, a *PublishChanges* must not be done while the tables are being updated by running application processes. See this diagram.



Let's assume at t1 process X is started, which will update table T. After that, at t2 a *PublishChanges* is done which creates or changes an audit profile including table T. However, process X will remain running with the audit settings as they were at t1. Consequently, as long as the Bshell is running the auditing will not be done as specified at t2.

And there is a more severe problem. Let's assume at t3 a new process Y is started. This process will load the new audit settings from t2. As soon as it does a transaction on table T, the audit trail is updated to reflect the new audit settings. (In technical terms: a new sequence file is initiated.)

So far so good. But if process X will now try to do an update on table T, the audit server detects that it is still working with old settings and the audit trail has switched to newer settings. This causes the transaction of process X to fail. Error 254 (commit transaction failed in the audit server) occurs.

Process Y and other processes started after t2 will not be impacted by this problem.

Performance

Regarding performance impact of auditing, see “Performance and Disk Space Usage”.

Timing

When synchronizing using *PublishList* and/or *PublishChanges*, it is important to take timing into account. The handling of related business objects and the coordination of *PublishList* and *PublishChanges* are already discussed in the “The Client Side”.

Additionally, note that the basic idea of the *PublishChanges* is that events are published as soon as they occur on the backend. This means that the load is spread over time. The other side to this is that multiple events on the same business object instance are not combined (net changed) into a single event.

Aside
When required, the middleware (in other words, the component outside LN that implements the actual publishing) could schedule the forwarding of event messages and combine events on the same object instance. Or alternatively, the client could lay up incoming messages and process them at a convenient moment. For scheduling, persistency is required, because multiple (potentially many) messages must be kept before sending them across the network.

If storing and scheduling is done, you can do net changing. In case multiple changes (transactions) were executed on the same object instance, they could be net changed, so the client must only process a single change. Note however that the rules for sequencing as described in “The Client Side” section do not apply anymore in that case, because a single net changing contains changes from multiple moments.

Multiple Clients

In case of multiple clients require receiving change events for the same business object from the same LN company, the requirements (selections and filters) of those clients must be merged before actually sending the *PublishChanges* request. The *PublishChanges* doesn’t merge the selections and/or filter with previous selections or filters. It is assumed that this is done outside LN. So a new *PublishChanges* invocation overwrites the settings of the previous one.

So one *PublishChanges* request is sent, and consequently the events are published to a single destination (bus component). A brokering solution is required to distribute the events to the clients involved.

Aside: these limitations can be avoided by duplicating the synchronization object using the Synchronization Server user interface (see “Troubleshooting”). However, this is not advised, because it causes multiple processes running on the backend for a single business object.

This table shows an example of merged selections and filters.

	Attribute Selections	Filters
Client X	A, B and C	B > 0
Client Y	A, D, E, F, G, H, I, J, K, L and M	K = “open” or K = “planned”
Resulting settings	A, B, C, D, E, F, G, H, I, J, K, L and M	if B and K are in the same component: (B > 0) or (K = “open” or K = “planned”) If B and K are in different components it is not possible to use a merged filter

Note that filtering cannot simply be done by juxtaposing the filters of the clients using either ‘and’ or ‘or’, because the components being filtered must be taken into account.

This means the selections must either be handled at the client site (which increases network traffic and processing time there), or a broker must be used to deliver to each client what is needed.

Reliability

Regarding guaranteed delivery and exception handling you must take this into account:

When publishing, the Synchronization Server does not use a persistent store for the events being published. The Synchronization Server keeps track of its status (what transactions were processed) but it does not store business object contents. So if required, the middle tier make sure messages are delivered to the client or messages are kept in a queue in case a target site is temporarily unavailable. This is not the responsibility of the LN backend.

No mechanism is offered for automatically retrying the processing or publishing of a change event. However, the server can be stopped if needed and started at a moment in the past to reprocess the same transaction. Alternatively, a *PublishList* can be used to make a fresh start.

In case of an exception the Synchronization Server will write its log file and either continue or stop. Stopping is done if the error is likely to be a generic problem rather than related to a specific business object instance.

Note however that the client will not automatically detect if a server has stopped. If no change events are arriving, the client won’t know whether this is because no relevant changes are taking place, or because a problem occurred.

It is strongly advised to create a job for regularly running the *Check and Continue Servers* (danch2210m000) session to make sure stopped or interrupted servers are automatically continued.

For more information on troubleshooting, see “Troubleshooting”.

Limitations

Text Handling

For *PublishChanges*, no text handling is available currently. The text content is not published, but text numbers can be published. Consequently, changes of the text content are not detected. However, changes on text number can be detected. In other words, the creation of a new text, the deletion of a text or the (un)linking of a text to the business object.

Constraints on selection and filter

The constraints on selection are defined in the section “Constraints on Components and Attributes”. The constraints on filtering are defined in the “Filter” section.

Old and new values in change events

In case of a change event (other than create or delete), the new values are included in the event message. In other words, the current values after the change. The previous values are not included.

This means that *outside* LN (for example in the middle tier or at the client site):

- Filtering is limited. If the attribute values indicate a change event is out of range, you don’t know whether the old value of that attribute was in range or not. Consequently, to be sure the message must usually be passed on as a delete event. Otherwise the client will not receive delete events for changes that make the object become out of range.
- No consistency check can be done at the client site to see whether the old values match the client’s state. However, if ownership is implemented properly this won’t be needed.
- If the client uses other attributes to identify an object or component instance than the identifying attributes in LN, the client may not be able to find the corresponding instance in its database for a change event.
- In case of event handling, the event may not be defined completely. For example, if an action must be taken if an ordered quantity *increases*, this cannot be determined based on the event from LN, because only the new quantity is available in the event message.

Retrieving a Single Object Image

If in exceptional cases it is needed to retrieve the object image for a single instance, this can be done using the standard *Show* method. This section contains a few examples and notes.

Using *Show* for Ad-hoc Synchronization

Let's for example assume that all production orders are transferred from LN to a client application daily. However, sometimes rush orders occur, which must be input in the current schedule as soon as possible. Synchronizing such a rush order cannot wait till the end of the day. In that case the *Show* method can be used. This means a single object is transferred ad hoc.

Note that using *Show* for this purpose involves some risks:

1. Synchronizing one object may result in multiple *Show* invocations, because of dependencies. For example, the rush order may require up to date items, BOMs (bill of materials) or routings.
2. During the next normal synchronization run, all changes (including the rush order) are synchronized. Consequently, the client application will receive changes that are in fact already processed.

To solve this, these options are available:

In this case, alternatives for using *Show* are:

- Increase the synchronization frequency.

  For example, instead of synchronizing every day, synchronize thirty minutes. This means that all orders are synchronized more frequently, not just the rush orders.

- Synchronize incidentally.

  This means the user initiates an intermediate synchronization run using *PublishList*.
  From a data consistency perspective, this is a good solution (just like the previous one).
  But do note that a complete incidental synchronization run will take more lead time than a single *Show*. For example, suppose all production orders are regularly synchronized every night and this process takes more than an hour. If a rush order arrives in the afternoon, then the synchronization may take an hour, which may be too long for the rush order.

Using *Show* in Relation to *PublishList*

Additionally, can be used when data for *PublishList* is incomplete. For *PublishChanges*, the right sequence and completeness of changed objects can be guaranteed, because each change is part of a database transaction having a unique transaction id and a so-called commit time. Consequently, if changes on Items and Orders are published, the client can sort the event messages by the using this information. However, for *PublishList* we cannot be completely sure that the data is consistent, because making a snapshot from the database takes time. That means that in exceptional cases, the set of objects may be incomplete.

For example, let's assume that within a single transaction an item is inserted and an order referring to that item is inserted. When reading items and orders from the database for a *PublishList*, the database queries can be executed as close as possible, but there will always be a minimal time difference. Even though if it is only a matter of milliseconds, this can result in reading the newly created order, without reading the newly created item.

If this is the case, the client application receiving the data will not be able to add the order to its database, because the item is missing. The client now has two options: skip the order, assuming

that the order and its item will both included in the next synchronization run, or retrieve the item data using the *Show* method.

The risk discussed here can be reduced by ‘swapping sequence’: first request the orders and then (after some time, as required) request the items. But when receiving the data at the client site first apply the items then apply the orders.

Using *Show* in Relation to *PublishChanges*

In case an event type was changed from *change* to *create* when filtering, the object may not be complete. Then *Show* can be used to retrieve the missing parts of the object.

For example, let's assume a filter is defined saying an order must have status *open*. An order is created having status *prepare* and two order lines are added to that order. The filter will remove the order from the *PublishChanges* output, because the order status is not equal to *open*. Now if the order status is changed from *prepare* to *open*, the filter will take care that the action is changed from *change* to *create* (because the client did not receive the order before). However, the order lines are not updated, so they are not included in the *PublishChanges* output.

In that case the order lines can be retrieved using the *Show* method.

Checking the Status

PublishList

If no result (exception) is returned by the *PublishList* method, the process is started. This process automatically ends after processing the data set. No decent method exists to stop a running *PublishList* process.

The status of a running *PublishList* process cannot be viewed, other than through the messages that are sent to the client. No logging is available for exceptions that occur during the data collection and the publishing of the list messages, except that in some cases the default LN logging mechanisms are used, such as the logging of the virtual machine (Bshell) in $BSE/log on the application server.

PublishChanges

After successfully issuing a *PublishChanges* request a continuous process is running to detect and publish change events.

As a user you may want to know, for example:

- What *PublishChanges* services are running, whether any processes are stopped or interrupted
- Whether a service is running for a specific business object and, if so, what attributes are included.

Chapter 4 Publishing Interface - Methods

4

These methods are available:

- *PublishList* - to publish the complete set of object instances for a business object that meet the specified selection and filter.
- *PublishChanges* - to switch on publishing of events on object instances for a business object that meet the specified selection and filter, or to change the specified selection and/or filter for a previous *SyncChanges*.
- *UnpublishChanges* - to stop publishing change events for a business object.

Just like any business object method, the publishing methods have method has these arguments:

- the request, containing the input for the method
- the response, containing the resulting data for the method, and any warnings if applicable
- the result, containing the error message(s) in case the method failed.

The selection and filter to be used in the *PublishList* and *PublishChanges* requests are described later.

The response argument is not used for the publishing methods, because the response is not sent immediately, but it is published asynchronously.

Method Details

PublishList

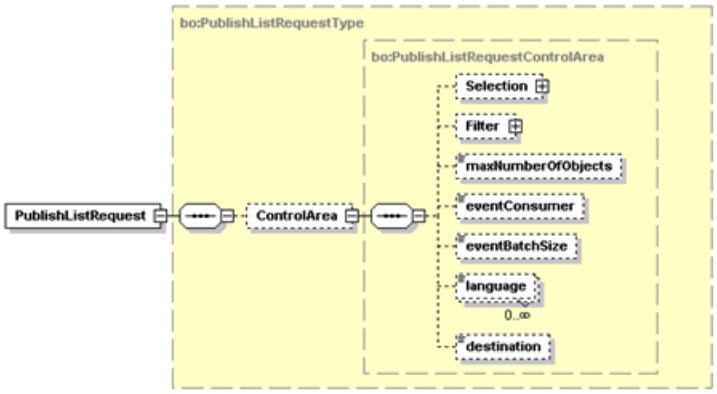
Using the *PublishList* method a request can be issued for a full list of a business object being published only once, to a specific consumer. The *PublishList* is in fact an asynchronous ListRequest. The response is not returned right away as in the synchronous ListRequest, but instead the response is being pushed out. This method publishes the complete data set based on the specified selection and filter.

The *PublishList* method is typically used at the start of a synchronization process or a reset of the synchronization process. Additionally, it can be used for business objects having relatively many

changes or relatively low data volumes, or with a very low synchronization frequency. This method is actually a replication of the complete data source at one side (LN) to the other side (client application), or, in other words, a complete regeneration of the data source at the other side.

Request

This diagram shows the *PublishList* request through which the business object is instructed to publish a list of instances. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements.



The *PublishList* request contains these controlling attributes:

- Selection

  What attributes must be included. The selection mechanism is equal to the *List* and *Show* method. If no selection or an empty selection is defined, all attributes are included.

- Filter

  What object or component instances must be included. The filtering mechanism is equal to the *List* and *Show* method. If no filter or an empty filter is defined, no filtering is done.

- maxNumberOfObjects

  Because a lot of data may be selected, the *PublishList* methods offer a mechanism to limit the total number of objects being published. A user can limit the total amount of instances to be pushed out by setting the maxNumberOfObjects attribute.

  The functionality is comparable to the maxNumberOfObjects functionality of the *List* request.

  If no maximum is specified, all object instances are published.

- eventConsumer

This is the name of the user that requests an asynchronous *List*. The name is included in the eventConsumer controlling attribute of the events that are being pushed out because of this request. Since event publishing (including the *List* event) is a broadcast mechanism, it is the responsibility of the client (or the hub) to filter out the specific events for that specific event consumer.

Note that change events published using *PublishChanges* are often for more than one client, but the data set as published using *PublishList* is usually for a single client.

If no eventConsumer is specified, the eventConsumer in the messages being published is empty.

- eventBatchSize

  Because an entire list of instances to be published can be rather large, they can be pushed out in batches. The maximum number of objects that should be pushed out in one message can be controlled by the eventBatchSize.

  If the *eventBatchSize* specified by the client is smaller than the amount of objects to be published, the application server publishes the instances in multiple messages of size equal to *eventBatchSize*. In the event, the controlling attribute eventHasMoreBatches indicates whether there are more batches to be published. If this attribute either does not exist or is set to false, the entire list is published.

  This controlling attribute is comparable to the *iteratorFetchSize* attribute of the *List* request. However, unlike for the *List* no new request has to be issued to get the next set of instances. A single *PublishList* is issued, which publishes one or more messages, depending on the eventBatchSize and the number of object instances available in the database.
  If this attribute does not exist or has been set to 0, the application server will publish the entire set of requested objects in one message.

  If no instance is in the database or in accordance with the specified filter, a single message having an empty data set is sent.

- language

  Clients should be capable of retrieving data from application servers and specify exactly in which languages they would like to see the text attributes of the instances. For LN, this feature is unused.

- destination

  The destination identifies the communication channel (bus component) to which the list needs to be published. The client must make sure an application that is able to process the incoming data is available at the specified destination (in other words: listening to the specified bus component).

  If empty or omitted, the default bus component is used. In that case, if no default bus component is specified, LN is not able to publish anything.

PublishChanges

Using the *PublishChanges* method a request can be issued for publishing changes of a business object. This method starts a service that publishes an event (create, change or delete) for a business object whenever it occurs, based on the specified selection and filter.

The *PublishChanges* method is typically used in a synchronization process where this method is more efficient than *List* or *PublishList* because only a relatively small portion of the data is changed. In that case, sending the change events is more efficient than resending the complete data set every hour or every day.

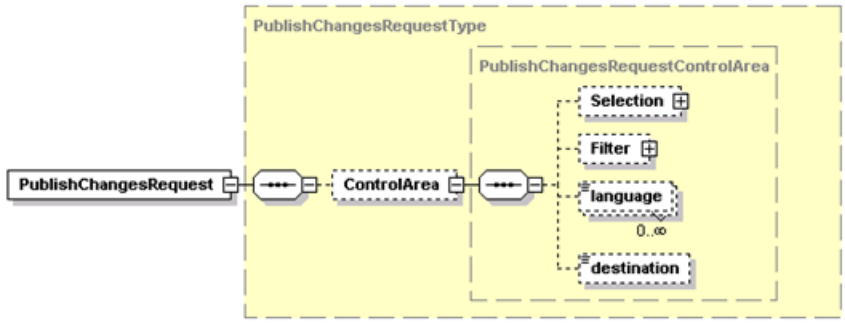
Changing the publishing behaviour (the Selection) for a specific business object can be done by issuing another *PublishChanges* request. Any subsequent *PublishChanges* request simply overrules a previous one, independent of the customer that has issued the request. For example, a client first requests the publishing of only changes for attributes A, B and C from a specific business object. Then in a second request a client requests only the attributes C and D, then the final selection will become C and D. It is not necessary to do an *UnpublishChanges* before another *PublishChanges* request in order to change the publishing behaviour. Merging of subsequent *PublishChanges* requests or *PublishChanges* requests done by multiple customers is not the responsibility of the LN application server and must therefore be handled outside LN.

When receiving a *PublishChanges* request, the Synchronization Server translates the publication requirements for a business object into a runtime publishing implementation consisting of a synchronization object and corresponding audit settings. The mapping from business object to synchronization object is implicit: the business object name is used as the synchronization object name. If the synchronization object already exists, that process is stopped and the existing synchronization object is changed. Otherwise a new synchronization object is created. The audit settings are updated if needed, and the service is started.

The immediate conversion of the audit settings to runtime imposes a risk. Running transactions for end users or application processes may be affected or even aborted.

Request

The diagram shows the *PublishChanges* request through which a business object is instructed to publish its changes. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements.



Controlling attributes

The *PublishChanges* request contains these controlling attributes:

- Selection

  What attributes must be included. The selection mechanism is comparable to the selection for the *List* and *Show* methods. However, in some respects the way the selection is used differs for change events.

  If no selection or an empty selection is defined, all attributes are included.

- Filter

  What object or component instances must be included. The filtering mechanism is comparable to the filtering for the *List* and *Show* methods. However, in some respects the way the selection is used differs for change events.

  If no filter or an empty filter is defined, no filtering is done.

- Language

  Clients should be capable of retrieving data from application servers and specify exactly in which languages they would like to see the text attributes of the instances. For LN, this feature is unused.

- Destination

  The destination identifies the communication channel (bus component) to which the change events must be published. The client must make sure an application that is able to process the incoming event messages is available at the specified destination (in other words: listening to the specified bus component).

  If empty or omitted, the default bus component is used. In that case, if no default bus component is specified, LN is not able to publish anything.

UnpublishChanges

With the *UnpublishChanges* an application server can be notified that publishing of the changes in the instances of a specific business object is not required anymore and should be stopped. The *UnpublishChanges* method does not have any parameters.

This method removes the publishing for the specified business object. Depending on the existence and status of the synchronization object, this method stops the service, deletes the synchronization object, and updates the audit settings.

The conversion of the audit settings to runtime imposes a risk. Running transactions for end users or application processes may be affected or even aborted.

Selection and Filter

Data sources must often only partly be synchronized. Not all information in the LN business objects is required for another application. Therefore, these specification is required:

- Which business object(s) must be synchronized.
- Which attributes of these business objects must be included (the selection).
- What instances of the business object must be included (the filter).

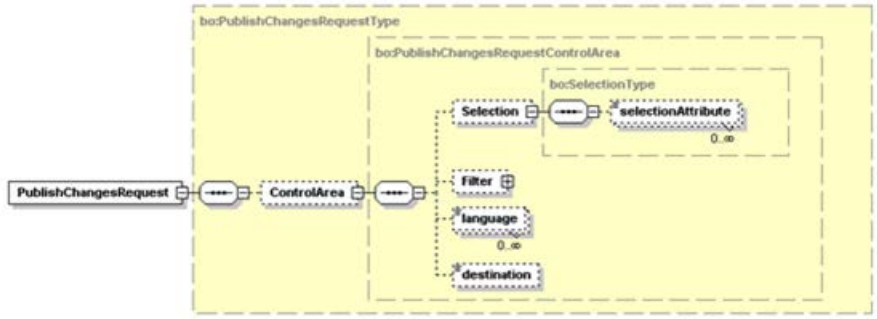
The business object is selected automatically, because a *PublishList* or *PublishChanges* is always invoked for a specific business object. The selection and filter are discussed below.

Selection

Selection is choosing specific components and attributes from an object, while leaving out others that are irrelevant. For example, fore an order only include the order number and date, and for the order line only include the line number, item, quantity and price attributes.

A selection can be specified by adding multiple selectionAttributes in the Selection element of the ControlArea element in the request. The way to do selections in the *PublishList* and *PublishChanges* request is equal to the way it is done in the *List* and *Show* request.

The diagram shows a graphical representation of the *PublishChanges* request XML schema definition with respect to the selection mechanism. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements.



Each SelectionAttribute element must contain an attribute that is listed in the request definition (XSD) as an attribute that can be included in the selection. Usually the list of attributes that can be selected is equal to the attributes that are available in the message definition (XSD).

In case the Selection element does not exist in the request or no selectionAttributes exist in Selection, it is assumed that *all* attributes as listed in the *PublishList* or *PublishChanges* message definition are requested.

The Synchronization Server uses a *different* selection format.

Example

In case only the order number and order date from an order, and the order line number, item, quantity and price for each order line must be included the selections is:

```
<ControlArea>
   <Selection>
      <SelectionAttribute> orderNumber </SelectionAttribute>
      <SelectionAttribute> orderDate </SelectionAttribute>
      <SelectionAttribute> lineNumber </SelectionAttribute>
      <SelectionAttribute> item </SelectionAttribute>
      <SelectionAttribute> quantity </SelectionAttribute>
      <SelectionAttribute> price </SelectionAttribute>
   </Selection>
</ControlArea>
```

How the selection is used for PublishChanges

For *PublishList* the selection will have the same outcome as for the *List* method. However, for *PublishChanges* it is handled differently, because change events are not the same as data as included in the *List* response. For change events, the selection doesn't mean all selected attributes are always included in every event message.

The selection is used in case an:

- Object or component is created, all selected attributes of that object or component are included.

- Object or component is deleted, only the identifying attributes of that object or component would be sufficient, but additional attributes may be included.
- Object or component is changed, at least the identifying attributes and the changed attributes are included.
- Aggregated child component (such as an order line) is created, changed or deleted, then also attributes from the parent component (the order) must be included.

See “Selection and Filter” for what components and attributes are included in an event message by default, and how these default settings can be changed.

If selection attributes are listed for *PublishChanges*, the selection must at least contain all identifying attributes for each component for which one or more attributes are selected.

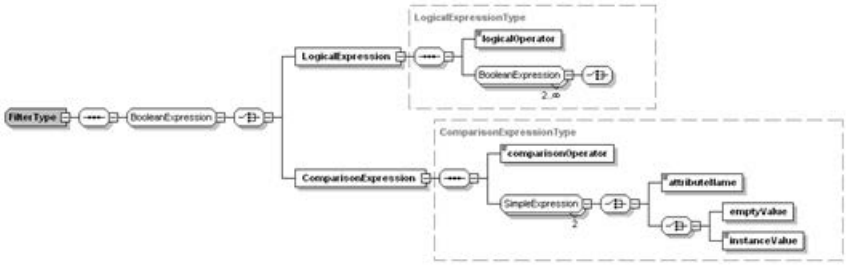
Filter

Filtering is choosing specific object instances or component instances, while leaving out others that are irrelevant. A filter is a specification of the conditions that must be met to include a particular object or component in the synchronization scope. For example, only includes orders having status 'released'. Or only include order lines having a planned date filled in. The filter will result in excluding irrelevant business object instances or component instances. A filter can be regarded as a performance optimization, because it avoids data being sent that is irrelevant for the client.

The filter in the *PublishList* and *PublishChanges* request is equal to the filter used for the *List* and *Show* methods.

A filter can be specified by adding a *ComparisonExpression* or by combining multiple *ComparisonExpression*s using a *LogicalExpression* in the *Filter* element of the *ControlArea* element in the request.

The diagram shows a graphical representation of the XML schema definition with respect to the filtering. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements. Rounded boxes are groups, which do not occur in the XML. Switches mean one of the listed elements is chosen.



These operators can be used:

Logical operators	Comparison operators	Numerical operators
or	le (less than or equal to)	/ (division)
and	lt (less than)	* (multiplication)
	ge (greater than or equal to)	- (subtraction)
	gt (greater than)	+ (addition)
	ne (unequal to)	
	eq (equal to)	

A value can be the actual value from the instance or it can be a tag called *emptyValue* which indicates that the value is empty. The element *emptyValue* contains no data.

If the Filter element is missing or empty (more specifically, no logicalExpressions or ComparisonExpressions are specified), no filtering is done, so all objects and components are included.

The Synchronization Server uses the same filter format.

Filtering components

Filters are defined per object, but they are applied per component, if a component instance exists. When defining a filter for a aggregated (multi-level) business object, this must be taken into account. If the attribute values are out of range, only the involved component instance (and its children, if any) is removed. Sibling components instances or parents are not impacted. If the top-level component is out of range this means the complete object is out of range.

For example, if an order object has two order lines, and a filter is specified for the order line component, then one order line may be excluded while the other one is included. But if a filter is defined on the order (header), either the whole order is included or excluded. Additionally, if due to a filter on the order line all order lines are excluded, this does *not* mean that the order (header) is excluded as well.

Using attribute values in filters

When setting the filter values, the format of the public attribute must be taken into account. This is especially relevant for:

- UTC attributes: do not define a filter such as *utcAttribute* > *1355346000*. A valid value for a UTC attribute is for example 2005-12-29T08:15:00Z.
- Enumerates: do not define a filter such as *enumAttribute* = *0* or *enumAttribute* = *txabcenum.value*.

If in the filter you must check whether a UTC value is empty or not, use an expression such as:

```
<ComparisonExpression>
    <comparisonOperator>ne</comparisonOperator>
    <attributeName>expiryDate</attributeName>
    <emptyValue/>
    </ComparisonExpression>
```

Enumerate values are also formatted using a standard method, but those should not have an empty value in the business object's persistent data. However, just in case, a filter using emptyValue (or an instanceValue containing an empty string) can be used.

To define a filter

The best approach is to define filter per component first, and then combine these using 'and'. Not that the 'and' in "orderNumber <> ‘999’ *and* orderDate is not empty" has a different meaning than "orderNumber <> ‘999’ *and* lineNumber <> 0", because the second one is about two components, so actually two filters!

This table shows an example:

Definition steps	Example
Step 1: Define the filter per component	Order header: (lifecycle=”approved”) or (lifecycle=”toBePlanned”) Order line: deliveryDate<”2004-01-01T00:00:00Z”
Step 2: Concatenate the filters per component using 'and'	((lifecycle=”approved”) or (lifecycle=”toBePlanned”)) and (deliveryDate<”2004-01-01T00:00:00Z”)
Step 3: Formulate the filter in the Reversed Polish Notation.	and (or (eq (lifecycle, ”approved”), eq (lifecycle, ”toBePlanned”)), le (deliveryDate, ”2004-01-01T00:00:00Z”))
Step 4: Construct the XML from left to right	See below.

The resulting XML is:

```
<LogicalExpression>
  <logicalOperator>and</logicalOperator>
  <LogicalExpression>
    <logicalOperator>or</logicalOperator>
    <ComparisonExpression>
      <comparisonOperator>eq</comparisonOperator>
      <attributeName>lifeCycle</attributeName>
      <instanceValue>approved</instanceValue>
    </ComparisonExpression>
    <ComparisonExpression>
      <comparisonOperator>eq</comparisonOperator>
      <attributeName>lifeCycle</attributeName>
      <instanceValue>toBePlanned</instanceValue>
    </ComparisonExpression>
  </LogicalExpression>
  <ComparisonExpression>
    <comparisonOperator>le</comparisonOperator>
    <attributeName>deliveryDate</attributeName>
    <instanceValue>2004-01-01T00:00:00Z</instanceValue>
  </ComparisonExpression>
</LogicalExpression>
```

Constraints and Limitations

Filtering is done on attribute values. These types of filters are currently *not* supported:

- Filtering on event type. For example, only publish newly created objects, or only publish changes on existing objects.

  However, in theory it is possible to only include create and delete events, by only selecting identifying attributes.

- Filtering based on the user of session (program) that executed the change in the LN backend.
- Filtering based on a state that is reached. For example, an event must be published when the status of an order *becomes* 'released', but no change events are needed if the status is released and some other selected attribute is changed. Another example: publish an event if the planned date *becomes* higher than the requested delivery date.

These types of filters are irrelevant in a change-based synchronization scenario, but they are relevant in an event-handling scenario.

Filtering must *not* be done on attributes that get their value from outside the business object (e.g. from a related business object), unless this data is static. For example, filtering on order lines on item type, if the item type is not from the order object, but from the item object.

This is important because a change of the item type is not handled as a change of the order business object. This means that a client synchronizing orders using *PublishChanges* may not have the current item type available until the moment a complete synchronization using *PublishList* is done.

Within the scope of a logical 'or' expression, attributes from only one component can be used. It is no use to define a filter like "Order.date is filled or OrderLine.date is filled".

Additional constraints are implied by the XML schema definition, such as:

- Because no numeric expression can be defined, you cannot define a filter such as: *quantity * price > thresholdAmount*. The same holds for string expressions.
- No 'between' is available. Instead, use 'and' in combination with '>' (or '>=') and '<' (or '<=').
- You cannot directly use a boolean attribute in the filter, such as *isPlanned*. Instead, use a comparison expression, such as *isPlanned = true*.
- You cannot use logical operator 'not', e.g. not(x = 3). Instead, invert the operation, e.g. *x <> 3*.
- You cannot define filters on complex types (such as an XML data type).
- You cannot define filters that aggregate multiple component instances, such as:
  - Only include orders having at least one order line.
  - Only include orders whether the total amount for all order lines exceeds $ 1,000.
- - You cannot use library functions or method invocations directly, e.g. a filter like tcibddll0001.must.be.planned.by.planner(item). Work-around: define an attribute 'plannedByPlanner' (= tcibddll0001.must.be.planned.by.planner(item)).

Filtering can result in incomplete event messages. For example, assume a filter is defined saying an order must have status *open*. An order is created having status *prepare* and two order lines are added to that order. The filter will remove the order from the *PublishChanges* output, because the order status is not equal to *open*. Now if the order status is changed from *prepare* to *open*, the filter will take care that the action is changed from *change* to *create* (because the client did not receive the order before). However, the order lines are not updated, so they are not included in the event message.

Result

If the return value of a method is unequal to zero, the result argument is filled. The result structure has a standard format, which is the same for all methods. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements.

Chapter 5 Publishing Interface – Messages

5

Messages

After requesting a *PublishList* or *PublishChanges*, the client receives messages.

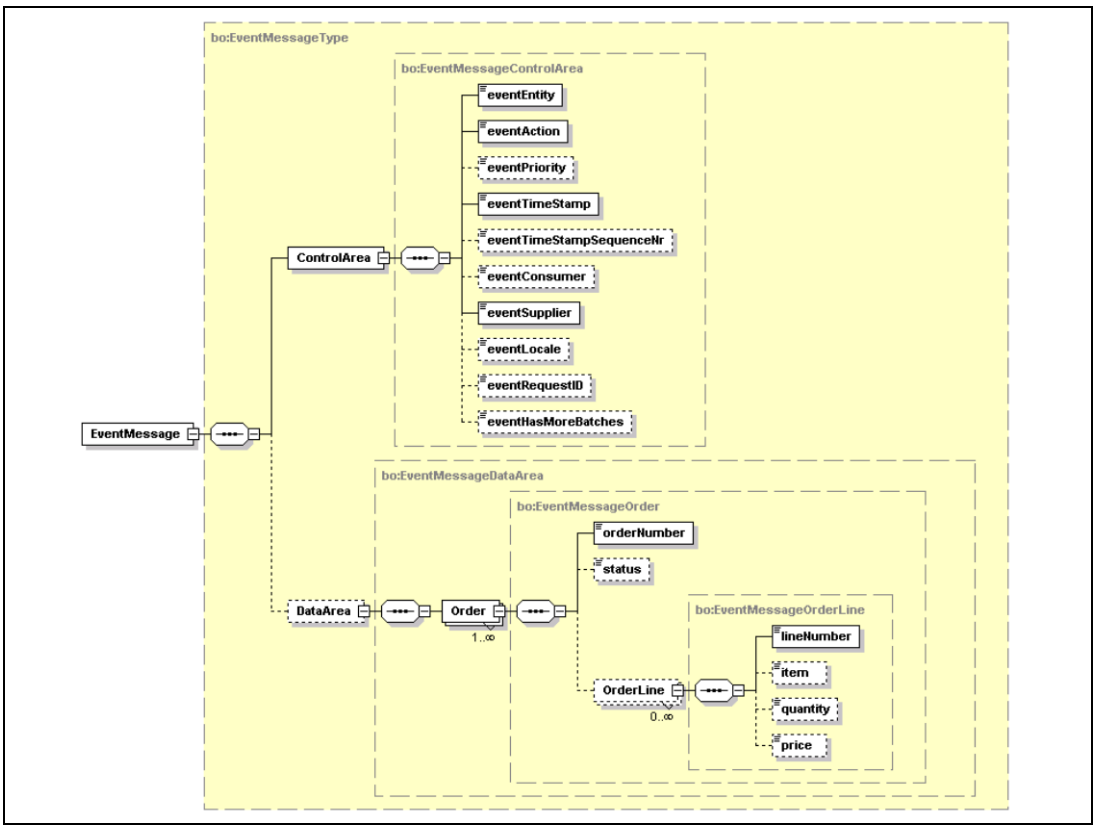
Message Definition

When publishing a list, multiple objects can be packed in a single message. When publishing change events, each object instance is published in a separate message. No bundling of multiple events in a single message (for example, per transaction) is done. The main reason is that a change event message cannot contain multiple instances, unless all events have the same action type (create, change or delete). But since a transaction can contain creates, changes, and deletes, we cannot assure the action types are equal for all objects being bundled. Additionally, the action can be changed during filtering for one or more instances, so even if the object instances in a transaction would initially have the same event action, this might not be the case anymore further on in the process.

In principle a create, change or delete event that is received by a client, can be used for issuing a create, change or delete request into the client's application server (although it can also be used in other ways). Therefore, the content of the XML based event message is aligned with the contents of change request message, enhanced with additional elements and attributes needed for event handling.

The XML document that describes the create, change or delete event is comparable to the request for a create, change or delete method. The XML document containing the published list is comparable to the response of a *List* request. A number of additional control elements are added for the event message.

This diagram shows the event related controlling attributes of an event message. Dotted boxes are optional XML elements; solid boxes are mandatory XML elements.



The data area shown here is only an example, because the content depends on the components and attributes of the business object involved. The example is based on a two-level business object consisting of order and lines.

Control Area

The event message contains these controlling attributes:

- eventEntity

  This is the name of the business object involved. For example: SalesOrder.

- eventAction

  This is the type of change that has occurred.

  In case of a message generated using *PublishList*, the eventAction is list.

  In case of a message generated using *PublishChanges*, the eventAction is create, change or delete. If the top-level component (and consequently the whole object) is created, the action type is create. If the top-level component (and consequently the whole object) is deleted, the action type is delete. In other cases, the action type is change. Note that in the latter case at least one component or subcomponent of the object instance is changed, create or deleted. This is indicated by the actionType attributes for the components in the data area.

- eventPriority

  This element is unused.

- eventTimeStamp

  In case of *PublishList*, this will contain the date/time at the moment the process started reading the business object instances in the LN backend. For database transactions, it is known as the 'commit time' in LN.

  In case of *PublishChanges*, this will contain the date/time at the moment the event occurred in the LN backend.

- eventTimeStampSequenceNr

  For *PublishChanges*, this element contains a sequence number for the transaction at the LN backend in which the event occurred. In case multiple events occur within a single second (so they have the same eventTimeStamp), this number identifies the sequence of the events. For database transactions, it is known as the 'transaction id' in LN.

  For *PublishList*, this element is unused.

- eventConsumer

  For *PublishChanges*, this element is unused.

  For *PublishList*, this element will contain the eventConsumer as specified in the request. This way the client can filter out relevant list events, because in case of *PublishList* the event messages are being pushed out for one specific client.

- eventSupplier

  Identifies the application server that published the event. It will define that the event came from LN and additionally the LN company number is included.

- eventLocale

  If available, this element identifies the locale that is being used for the data inside the event. The locale is a string, in which languages can be specified according to the RFC-1766 standard (<language code-country code>, please (see http://www.ietf.org/rfc/rfc1766.txt).

  The format is language code-country code. An example of such a locale is 'en-us'.

  The *language* part is a valid lower-case two-character language code as defined by ISO 639 (see for example http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt).

  The *country* part is a valid upper-case two-character country code as defined by ISO-3166 (see for example http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html).

- eventHasMoreBatches

  This element indicates whether there more batches are to be received. In case *PublishList* is used and the application server pushes the data set out in batches (for example, because an eventBatchSize was set in the *PublishList* request) then this flag is relevant. The value is true if the message is not the last one for the data set to be published. Otherwise the value is false or the element is omitted.

  For *PublishChanges*, this element is unused.

Data Area

For a *PublishList* message, the data area is equal to the data area of the List response.

For a *PublishChanges* message, the data area depends on the event action:

- For create events, the data area is equal to the data are of the create request.
- For delete events, the data area is equal to the data are of the delete request.
- For change events, the data area is comparable to the data are of the change request. Only change events from *PublishChanges* may contain more unchanged attribute values than a normal change request requires, and additionally the actionType attribute may be included more often.

Aside: the data area used internally in Synchronization Server has additional elements, such as a PreviousValues element for the old attribute values, and some additional xml attributes. This is visible only when tracing. However, before publishing the data area is aligned to the standard create, change and delete methods.

For a *PublishList* message, all component instances are included, unless the filter considers them to be out of range. For each component instance, all selected attributes are included.

For a *PublishChanges* message, unchanged component instances and unchanged attribute values may not be included.

Caution:

Although the both message definition (as stored in the XSD) for the *PublishChanges* and the selection from the request specify a sequence of the attributes within a component, this sequence is not guaranteed by the Synchronization Server.
Additionally, the sequence of the component instances within their parent component is undefined. So the order lines within an order object are not ordered by order line number, for example.

Example

This is an event in which three changes are made an Order. The quantity of OrderLine 1 is changed, a new OrderLine 3 is created, and the existing OrderLine 2 is deleted.

```
<EventMessage>
    <ControlArea>
        <eventEntity>Order</eventEntity>
        <eventAction>change</eventAction>
        <eventTimeStamp>2004-01-19T10:56:38Z</eventTimeStamp>
        <eventTimeStampSequenceNr>1849578429374576534</eventTimeStampSequenceNr>
        <eventSupplier>ERP LN 590</eventSupplier>
    </ControlArea>
    <DataArea>
        <Order actionType="unchanged">
            <orderNumber>ORDER001</orderNumber>
            <Line actionType="change">
                <lineNumber>1</lineNumber>
                <quantity>100</quantity>
            </Line>
            <Line actionType="create">
                <lineNumber>3</lineNumber>
                <item>ITEM3</item>
                <quantity>200</quantity>
                <price>15.00</price>
            </Line>
            <Line actionType="delete">
                <lineNumber>2</lineNumber>
            </Line>
        </Order>
    </DataArea>
</EventMessage>
```

Remember that, equal to the change method, creation and deletion of subcomponents are not considered to be create events or delete events, but change events for the business object.

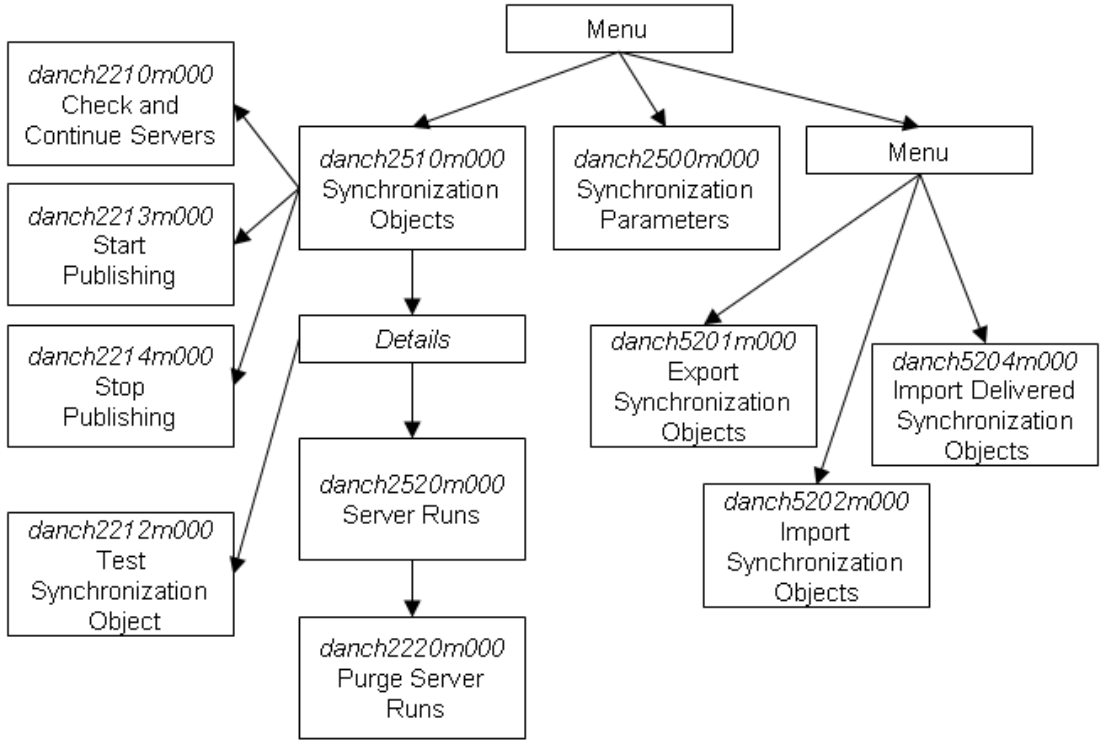
Chapter 6 Synchronization Server Setup and Deployment

6

By default, a synchronization object is created from the outside world by invoking the *PublishChanges* method. Alternatively, the Synchronization Server user interface can be used to create or fine-tune a synchronization object.

User Interface Overview

The sessions and their relations are shown in the diagram. An arrow indicates that a session can be started from a menu or a command in another session. Print sessions are not displayed in the diagram.



For detailed information, see the online help for the individual sessions.

User Tasks

A number of typical user tasks are listed here. For each task, the sessions/commands to be used are mentioned.

Define a new synchronization object

In the *Synchronization Objects* (danch2510m000) session, you can define a synchronization object. Then create the runtime using the command in that session.

Normally this task is not done via the Synchronization Server user interface, because a synchronization object is created automatically when a *PublishChanges* request arrives for a business object.

Note:

Before being able to start publishing, the audit profile must be converted to runtime. This is an application administrator's task, which may impact end users of LN. The session to be used is Create Runtime Audit Definitions (ttaud3200s000), which can be started from the Audit Profiles (ttaud3110m000) session.

After setup and convert to runtime, run a test using the command from the Synchronization Objects (danch2510m000) session. Events are written to a file and the contents of that file is displayed after stopping the server.

Start publishing

In the Synchronization Objects (danch2510m000) details session, execute the ‘Start’ command.

Normally this task is not done via the Synchronization Server user interface, because a publishing is started automatically when a *PublishChanges* request arrives for a business object.

If a range of servers must be started and/or a specific starting moment is required, the Start Publishing (danch2213m000) session can be started from the Synchronization Objects (danch2510m000) overview session.

A specific starting moment is required if:

- The server didn’t run before and should start at some moment in the past instead of the current moment. E.g. publish all events using transactions executed since the beginning of this week.
- The server ran before but has a large backlog. For example, if it has not been running for a month then it could be started at the current time, instead of continuing where it stopped (provided the clients for the change events do not need the events from that month anymore).
- A range of servers must be started at the same point in time, for example, because the business objects are related and consequently the change events may be interdependent.

View Status and Solve Issues

In the *Synchronization Objects* (danch2510m000) details session, view the status (whether the server is running or not). If needed, the history can be viewed by starting the *Server Runs* (danch2520m000) session. Using that session, also the log file for each run can be viewed, which contains the exceptions that occurred.

If required, the complete status of the synchronization object can be reset in the *Synchronization Objects* (danch2510m000) details session.

For details on troubleshooting see “Troubleshooting”.

Change a synchronization object

The user changes the settings for an existing synchronization object using the *Synchronization Objects* (danch2510m000) details session. After that, if the business object, selection or filter was changed, the configuration library and audit profile are regenerated. Finally, if a server is running for processing change events, that server must be stopped and started again (continued) to make use of the new settings.

For details, see “Tuning”.

Ensure servers keep running

Add the *Check and Continue Servers* (danch2210m000) session to a job that is executed according to a predefined interval or calendar.

Cleanup

To clear the history, use the Purge Server Runs (danch2220m000) session, which can be reached from the Synchronization Objects details session using the History command and then selecting Purge from the Server Runs session. Do not clear the last run, unless you want to reset the current status. After clearing the last run, the server cannot continue anymore from the point where it stopped last time.

Reset the synchronization object (including the current status) can also be done using the Reset command in the *Synchronization Objects* (danch2510m000) details session. In that case all history and status data is removed.

Cleanup for the Audit Trail is done using the Audit Management sessions.

For the Synchronization Server no transaction data needs to be cleaned up, because after an event is passed on to the publication channel (bus component) it is not available anymore in the Synchronization Server.

See also “Performance and Disk Space Usage” on disk space usage.

Stop publishing

In the Synchronization Objects (danch2510m000) details session, run the ‘Stop’ command.

If a range of servers must be stopped, the Stop Publishing (danch2214m000) session can be started from the Synchronization Objects (danch2510m000) overview session.

The publishing is stopped automatically if an *UnpublishChanges* request arrives for the business object.

Delete a synchronization object

If the server is running, first stop it using the Synchronization Objects (danch2510m000) details session. Then delete the synchronization object using the overview session.

The synchronization object is deleted automatically if an *UnpublishChanges* request arrives for the business object.

From *PublishChanges* to a Synchronization Object

Relation between Publishing Methods and Synchronization Objects

Upon the first *PublishChanges* for a business object a synchronization object is created. For subsequent *PublishChanges* invocations for the same business object, the settings (selection and filter) for the synchronization object are overwritten. If an *UnpublishChanges* arrives, the corresponding synchronization object is deleted.

The *PublishChanges* and *UnpublishChanges* methods do not only impact the Synchronization Server, but they also impact the audit settings. See also “Audit”.

When a new synchronization object is created, a number of default settings are used. This includes:

- The synchronization object name equals to the business object name.
- The behaviour of the selection for a synchronization object can be defined more specifically than in the selection of the *PublishChanges* method. This is explained in “Selection and Filter”.
- The location where the log file is stored is determined. This is done based on a parameter from the *Parameters* (danch2500m000) session. We recommend that you check the value for this parameter and update it as required. The log file can be viewed via the History command from the *Synchronization Objects* (danch2510m000) session.
- A number of other attributes as specified in the *Synchronization Objects* (danch2510m000) session get a default value. See “Configure the Synchronization Object” and the online session help for detailed information.

The bus component (destination) as defined in a *PublishChanges* request is not stored for a synchronization object. For the Synchronization Server, the bus component (destination) is a runtime aspect. It is not in the configuration, but is set when starting the server. If desired, the user can switch to another bus component by stopping the server and continuing it while specifying the other bus component.

When the *PublishChanges* method is used successfully, the synchronization object is created and the server is started using the specified bus component (taken from the 'destination' element of the *PublishChanges* request).

When the *PublishChanges* is unsuccessful, after solving the problem the user can either start the publishing from the Synchronization Server user interface (but in that case the user must specify the bus component), or invoke the *PublishChanges* method again (which again includes the bus component in the destination element).

Checking the Status

In LN, sessions are available to view (and change) the *PublishChanges* settings. The user can check what synchronization objects are active, for what business objects and what attributes, and what their status is.

Currently, no public business object methods are available for checking the event publishing status for a business object.

This table explains how to complete a number of tasks:

What	How to
See what PublishChanges services are defined, or check whether a service is defined for a specific business object	Ensure your current LN company is the company you want to view. Start the Synchronization Objects (danch2510m00) session. Synchronization objects having the same name as a business object are created from outside LN by the PublishChanges request. You can also print the available synchronization objects using the Print Synchronization Objects (danch2410m00) session, which can be started from the Synchronization Objects (danch2510m00) session
Check what attributes are included for a business object	From the Synchronization Objects (danch2510m00) session, view the details of a synchronization object. The business object components and attributes are listed in the Selection text of the synchronization object. For more information, see “Selection and Filter”.
Check what PublishChanges services are running and whether any processes are stopped or interrupted	From the Synchronization Objects (danch2510m00) session, view the details of a synchronization object. The status information is available on the Status tab. For more information, see the online help of the

danch2510m00 session.

Tuning

Based on the settings from the Business Object Repository, the Synchronization Server is configured. The *PublishChanges* request defines what business objects can be synchronized and what components and attributes are relevant, and which object and component instances are relevant. Based in this input, a synchronization object is created.

After a *PublishChanges* is executed, the settings of the generated synchronization object can be changed as required.

Changes can be done in two categories:

- Tuning the configuration of the events to be published (selections and filter). For example, to increase performance.
- Configuring other settings.

After changing the configuration, run a create runtime. This is discussed in “Create Runtime”.

For an overview of Synchronization Server sessions, see “User Interface Overview” or the online help.

Selection and Filter

The selection and filter can be changed for a synchronization object. In general, it is not required to change the filter, because the correct filter can be set directly in the *PublishChanges* request. For the same reason it is not required to add or remove attributes to or from the selection.

However, the ‘*includeUnchanged*’ settings in the selection can be changed. The reasons for doing this are:

- To increase performance and reduce load on the LN system, we recommend that you reduce the unchanged data being included. Preferably, *includeUnchanged* must be off for all components and all attributes. When processing an event, unchanged data is read from the database and merged into the event message. This significantly impacts performance. Consequently, the *includeUnchanged* should only be one in cases where it is really required for the client application.
- To include additional unchanged data. By default, unchanged child components are not included. For example, if an order line is changed, only that order line and the order header are in the event message. But, if a client really requires receiving *all* order lines (so also the unchanged ones), the *includeUnchanged* can be switched on for the OrderLine component.

Changes to the selection or filter are made using the *Synchronization Objects* (danch2510m000) session.

Caution:

Changes to the selection or filter of a synchronization object that was created using the *PublishChanges* method are overwritten by a new invocation of *PublishChanges* for the same business object!

Configure the Synchronization Object

For the synchronization object, the user must specify:

- What buffer size, polling frequency and log folder (directory) to use. See the online help information for details.
- The folder (directory) to contain the log data.
- Optionally, any specific settings for tracing, debugging or profiling. These are discussed in “Troubleshooting”. If no problems arise it is not required to check these settings.

  You can use the *Synchronization Objects* (danch2510m000) session.

  Note that each synchronization object additionally has a reference to a configuration library and an audit profile. These are generated based on the definition of the synchronization object (see “Create Runtime”).

Create Runtime

Before you can use a created or changed synchronization object, a ‘create runtime’ must be done. This means a configuration library and an audit profile are created.

Configuration Library

For each synchronization object, a configuration library must be created. This library contains the specific code to be executed. It defines the meta data (that is, the object structure including components and attributes). Both the 'inside' (related to database tables) and the 'outside' (the final result as received by the client application) are modelled. Additionally, transformation and filtering logic are implemented in accordance with the business object implementation and the selection and filter as defined for the synchronization object.

The configuration library is generated from the *Synchronization Objects* (danch2510m000) session. The configuration library has the same package VRC as the synchronization object that is used. Consequently, it is automatically available in the used package combination, because the business object is also available in the package combination.

Note that the existing configuration library (if any) is overwritten. No check is done whether the library was adapted after the previous create runtime.

Audit Profile

Upon create runtime, an audit profile is also generated for the synchronization object. This profile ensures that the required data is audited in the audit trail.

After generating an audit profile, the audit profile must also be converted to runtime. This is an application administrator's task. Use the *Create Runtime Audit Definitions* (ttaud3200s000) session. You can start this session from the *Audit Profiles* session (ttaud3110m000) session. For details, see the technical manual.

Caution:

Just creating or changing the audit profile (when creating the runtime for a synchronization object) does not impact end users. But do not create the runtime audit definitions while user transactions are being executed.

Notes

When creating the runtime:

- Runtime information (such as server runs) or the configuration settings for the synchronization object are not changed.
- If a server is running, the server must be stopped and continued before the changes become actual. This is not done automatically. Do not forget to create the runtime audit definitions before starting and continuing the server.

‘Manually’ Setting up a Synchronization Object

The Synchronization Server user interface can be used to create a new synchronization object ‘from scratch’. This option should only be considered in situations where the *PublishChanges* method cannot be used.

This section discusses how to:

- Create a synchronization object
- Test and deploy the synchronization object

Definition

A synchronization object is created in the *Synchronization Objects* (danch2510m000) session. Either an existing object can be duplicated, or new one can be created. In the exceptional case that synchronization objects are delivered with LN or an integration pack, they can be imported. Use the *Import Delivered Configurations* (danch5204m000) session.

If multiple clients (sites or applications) exist that must synchronize data based on change events for the same business object, it is worthwhile to investigate whether the same synchronization object can be used. It reduces the number of processes that are collecting and processing changes.

When defining a synchronization object, the most important steps are:

- Selecting the business object.
- Entering the selection XML
- Optionally, entering the filter XML

After that the other configuration settings for the synchronization object are defined.

Note:

Constraints as specified for publishing methods are also relevant when ‘manually’ creating a synchronization object. Additionally, see “Constraints on Components and Attributes” on what components and attributes can be used.

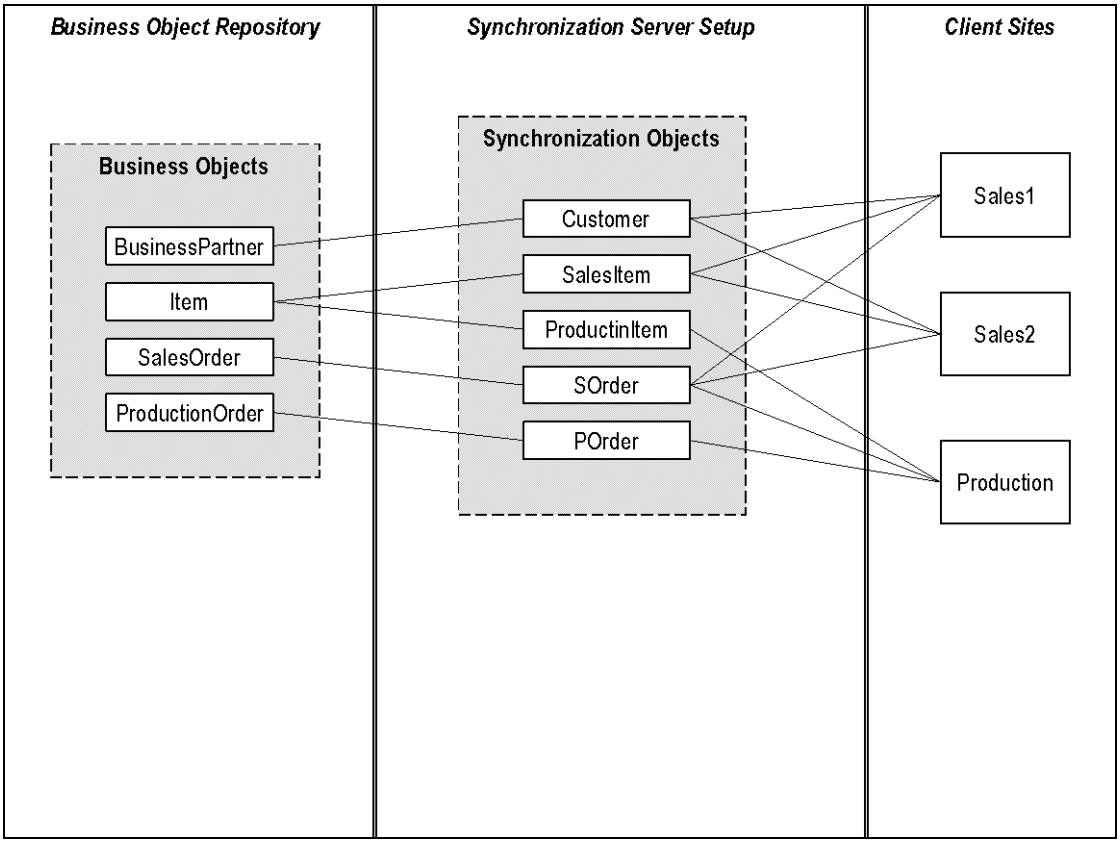
Example

This example illustrates the relations between the client sites, the synchronization objects and the business objects from the Business Object Repository.

In this example, three client sites exist. The two sales sites have the same integration requirements. The third site is a production plant, which has different requirements for integrating with LN.

The sales sites need data for Items, Sales Orders and Business Partners. They use the SalesItem synchronization object for the Item object, the SOrder synchronization object for the SalesOrder object, and the Customer synchronization object for the BusinessPartner object.
The production site needs data for Items, Sales Orders and Production Orders. It uses the ProductionItem, SOrder and POrder.

The diagram shows the resulting synchronization objects. Note that the Synchronization Server publishes each event once for each synchronization object. If it must be distributed to multiple client sites this must be done in the middle tier (for example, using a broker).



As mentioned before, it is worthwhile to try to combine the SalesItem and ProductionItem in a single synchronization object.

Testing and Deployment

A test can be executed by doing two things in parallel:

- Run the synchronization server process.
- Execute transactions on the tables involved.

To do this locally without testing the complete integration, the *Test Synchronization Object* (danch2212m000) session can be used.

A simple test procedure:

- Create a base data set for the business object, for example by entering data through the application session.
- Run the *Test Synchronization Object* (danch2212m000) session.
- When prompted, make some changes to the data set (adding, changing or deleting business object instances).
- Check the results as displayed by the *Test Synchronization Object* (danch2212m000) session.

If the outcome is not as expected, check the setup of the synchronization object (and the business object in the Business Object Repository, if required). Additionally, the information in “Troubleshooting” can help to find the cause of the problem.

Take into Production

Probably, integrations are initially implemented in a test or shadow environment. After completing the development there, the configurations must be duplicated to production environments, which may be different LN companies or different systems.

To facilitate this, Synchronization Server configurations can be exported and imported. Use the *Export Synchronization Server Configurations* (danch5201m000) session to export the configurations to a file. Use the *Import Synchronization Server Configurations* (danch5202m000) session to import the configuration from the file.

The export of configurations does not include the bus component, because that is a runtime aspect for the Synchronization Server. This means it must either be specified in the *PublishChanges* method, or it must be entered when starting a synchronization object via the user interface.

Important

The export and import is not needed if the event publishing on the production site is set up using the *PublishChanges* method.

Selection and Filter

Selection

Caution:

The selection as defined for a synchronization object differs from the selection as used in the *PublishList*, *PublishChanges*, *List* and *Show* methods. The synchronization object selection contains more information and allows the user to fine-tune the publishing of change events.

Selecting the components and attributes to be included is done by attaching an XML text to the synchronization object. Use the *Synchronization Objects* (danch2510m000) session. If a synchronization object is created automatically by the *PublishChanges* method, the selection text is generated.

This selection text specifies what business object components and attributes must be included, and which components and attributes must also be included if they are unchanged.

The attributes must be defined as public attributes in the Business Object Repository. In the Business Object Repository, the public layer doesn't have components defined explicitly, but still the component is defined implicitly in the Business Object Repository for each public attribute.

Additionally, components are defined in XSDs that are linked to the method arguments. In the Synchronization Server, the components are defined explicitly in the selection XML.

Regarding the components and attributes that can or cannot be included, see “Publishing Development Guide”.

Example

The selection text contains XML structures. The format of the selection text is (as an example the text of the regression test business object is depicted here):

```
<Order>
     <orderNumber key="true"/>
     <orderStatus includeUnchanged="true"/>
     <orderDate/>
     <OrderLine>
          <lineNumber key="true"/>
          <item includeUnchanged="true"/>
          <quantity/>
          <price/>
     </OrderLine>
</Order>
```

The root node holds the name of the root component, which has the business object name or the main component name (or indeed any other name). All nodes having children are components; nodes that do not have children are attributes belonging to the component that is their parent. Components having a parent are child components of their parent component. The components (except the top-level component) and attributes must exist within the business object, as specified within the Business Object Repository. Also the components’ parent-child relations must match the business object structure as defined in the Business Object Repository.

Note

In the Business Object Repository, a business object without any component can be defined. In that case, we regard the business object itself as the top-level component. In the selection we will have a component defined anyway.

For all components and attributes additional settings can be specified:

- *key*: whether the attribute is within the key. For each component that is included, *all* identifying attributes must be selected, and each of these must have the ‘key’ attribute set.
- *includeUnchanged*: whether the component or attribute must always be included (true) or only if it has been changed (false). Default value if omitted is true for the root component and the key attributes and false otherwise.

Unchanged Components and Attributes

In case of a change event, only the key attributes and changed attributes are included. But if *includeUnchanged* is set to ‘true’ then the attribute value is also included if it is unchanged. However, an attribute value is only available if the component instance is available.

You can include unchanged business object components. This means that together with the changes in another component, also the values of the key and *includeUnchanged* attributes of the component instances are included. If a child component is changed, the parent component(s) are included anyway, even if they do not have *includeUnchanged* set.

Setting *includeUnchanged* does not have the same meaning as using optional or mandatory attributes in the response XSD for another method, as explained in this table:

	Event publishing (set using ‘includeUnchanged’)	**Other methods (set using ‘minOccurs’)**
Mandatory attribute	Attribute is always available for a component instance in the object, also if the attribute's value is unchanged	Attribute is always available for a component instance in the object
Optional attribute	Attribute is available in a component instance if the attribute's value is changed	Attribute may be missing for a component instance

Regarding the *includeUnchanged* setting, take this into account.

For components:

- If a child component has *includeUnchanged* set, its parent component also must have *includeUnchanged* set.
- The setting includeUnchanged does not make a difference for the top-level component. After all, if an object instance is available then it must have an instance for the top-level component.
- We recommend that you unset *includeUnchanged* for components (other than the top-level component). Do not set *includeUnchanged* to ‘true’ unless really required.

For attributes:

- There is no relation between the setting of *includeUnchanged* for the attribute and the setting for the components.
- Setting *includeUnchanged* only influences unchanged attributes for changed components. It doesn’t influence the occurrence of attribute values in created or deleted components.
- We recommend that you unset *includeUnchanged* for all attributes. Do not set *includeUnchanged* to ‘true’ unless really needed.

  Using *includeUnchanged* is very expensive from a performance point-of-view. At runtime, the synchronization server has to do additional database read actions, and has to synchronize changes and additional unchanged data.

  Components or attributes should only have *includeUnchanged* set, if the client application cannot properly handle the change events otherwise. For example, if the client application identifies an object using different attributes than the ones regarded as key attributes by LN, the values for those attributes must be available.

Attributes that do not have *includeUnchanged* set can still be included in an event message event if their value is unchanged. This can have two reasons:

- Another audit profile lists the related table fields as fields to be always logged in the audit trail.
- The fields are required due to transformations or filtering.

Transformations or filters can require attributes or components to be included even if they were unchanged. However, this need not be taken into account in the selection of a synchronization object, because it is handled automatically when creating the runtime for a synchronization object.

The synchronization server may regard optional components or attributes as mandatory if it is required for:

- The filtering. When filtering on an attribute it must be available. For example, let's assume a synchronization object exists for an order object consisting of header and lines. If a filter is defined stating the status in the order must be 'open', we must make sure the status always is available. If includeUnchanged is not set in the selection of the synchronization object, it is handled at runtime automatically.
- Transforming the data from the database into the public components and attributes of the business object. For example, when combining two table fields into a number of new public attributes, both values must be available. And in case of transformations that go beyond the scope of one component, the Synchronization Server may need to regard components as mandatory.

Default settings

When using the *PublishChanges* method the selection is generated. In other words, when creating or changing a synchronization object based on a *PublishChanges* request, the selection as specified in “Selection” is translated to the selection as used in the synchronization object.

In that case:

- The component structure and ‘key’ settings are defined based on the business object definitions in the Business Object Repository.
- The includeUnchanged settings are by default off for all components and on for all attributes. The values of includeUnchanged cannot be changed via the *PublishChanges* method. However, the default settings can be changed using the Synchronization Server user interface (see “Tuning”).

  By default, not only the changed attributes are published but also the other selected attributes of the component in which the change occurs. Even if they were not affected by the change.

Consequently, the default behaviour for event publishing is:

1. All changed (or created or deleted) components plus all their parent components.
2. All attributes for each component being included.
3. In case of delete, the message can (but need not) exclude deleted child components of a deleted parent component and can (but need not) exclude non-identifying attributes.

4 In case an aggregated child component (such as an order line) is created, changed or deleted, then also attributes from the parent component (the order) are included. This is implied by 1 and 2.

Filter

To selecting the object or component *instances* to be included, attach an XML text to the synchronization object. This is done using the *Synchronization Objects* (danch2510m000) session. If a synchronization object is created automatically by the *PublishChanges* method, the filter text is generated.

Performance and Disk Space Usage

It is important is to take performance into account from the beginning. Additionally, based on the test results specific optimizations may be possible. A number of hints are described here.

PublishChanges versus (Publish)List

When synchronizing data, the List or *PublishList* method can be used to retrieve the complete data set. *PublishChanges* must be used if:

- Changes must be received by the client application quickly. In other words, if the allowed latency is small.
- The complete data set is large and the number of changes is (relatively!) small.

  For example, if the allowed latency is 1 day and each day 90% of all object instances is changed each day, it is cheaper to use the *List* or *PublishList* once every day to retrieve the complete data set. Or if only 50 object instances exist for a business object it may be cheaper to invoke the *List* method every fifteen minutes than to use *PublishChanges*.

Server Load

When using the *PublishChanges* method, be careful not to unnecessarily create additional synchronization objects for the same business object. Reuse existing synchronization objects for as many clients as possible. Synchronization objects should not be duplicated for multiple clients, unless the advantages outweigh the resulting performance cost. In case of minor differences between two synchronization objects, it is better to use one synchronization object that combines the requirements from both clients.

Adding a filter to a synchronization object reduces the number of instances included. Consequently, the network load and processing for the client is less. On the other hand, note that filtering can require attributes to be included even if they were unchanged, which can imply additional database actions. For example, when filtering on an order status from an order header, the order status may be included if a change occurs on one or more order lines.

Server load for auditing

Auditing has an impact on OLTP. If the environment is set up properly, the overhead should not be more than a few percent. If a very heavy auditing is done (auditing almost the complete database), an overhead of 10 to 15% must be expected. Note however that these figures very heavily depend on the situation (setup of hardware, network, number of users, etc.) it is very difficult to give general figures.

Even though the overhead is limited, the result can be significant if it is not taken into account when sizing. For example, if sizing is done in such a way that system resources are used up to 95%, users may hardly notice a 3% overhead, but a 6% overhead will bring down system performance to an unworkable level for all users.

Synchronization Object Configuration

The configuration of the synchronization server impacts performance.

The most important aspect is the *includeUnchanged* setting in the selection; this setting can be quite expensive from a performance point-of-view. Consider reducing the amount of unchanged data in event messages wherever possible.

Also including attributes from other business objects is expensive and must be avoided if it is not absolutely required.

In general, filtering increases performance, because less or smaller objects are sent across the network and less processing is required for the client application. We recommend that you use a filter that excludes all object or component instances that are not required.

Finally, tracing must be switched off unless it is required for troubleshooting (see “Troubleshooting”).

Business Object Configuration

The implementation of the business object also influences performance. Note that this is a development aspect. The business object implementation cannot easily be changed. However, it may be possible to exclude ‘expensive’ components or attributes from the selection if they are not absolutely required.

Disk Space Usage

Synchronization may involve a lot of data being transferred and stored.

Data is stored for the audit trail. Make sure sufficient space is available for files and table ttaud110. Note that by default the Transaction Notifications (ttaud110) for all companies are stored in company 000. Additionally, ensure that no bottlenecks occur for audit trail files. In general, we do not recommend that you use the $BSE directory. This directory usually already has a lot of input/output. Consider using separate disks and/or disk striping.

Purging audit data is done using the *Purge Audit Files* (ttaad4261m000) session. Note that the same audit data may be used for multiple purposes (for example, Synchronization Server, Exchange schemes, or EDP auditing in general). Do not purge data that must be retained; some data is audited for legal purposes. The use of audit data can be checked through the audit profiles defined in the corresponding audit management session. Also the *Where Used Audit Tables* (ttaud3521m000) session can be useful in this case. If the audit data for a table is not required anymore for any of the audit profiles that include that table, it can be purged.

The Synchronization Server does not store any business object data. Only its status and the log or trace data (if required) is stored.

Troubleshooting

This section contains information that is useful when handling problems.

This is specified:

- An overview of the available error handling in the Synchronization Server.
- How to handle exceptions that may occur when using the *PublishChanges* and *UnpublishChanges* methods or using the Synchronization Server user interface.
- Some notes on how to test a synchronization object and how to influence its status if required.
- A description of the server log file that should be checked when problems occur when running a server and the additional tracing the Synchronization Server offers.
- How to debug or profile a synchronization object.

Note:

In the Synchronization Server, a number of sessions are available to give information on synchronization objects, their status and their history. These sessions can be used for example to determine the location of log files, to check what servers are running or to view the log file.

Error Handling Functionality

Error Handling in PublishChanges

If the *PublishChanges* (or *UnpublishChanges*) method fails, a so-called ‘result’ structure is provided explaining what went wrong.

When creating the synchronization object based on the *PublishChanges* settings, the basic settings are checked. For example, is the specified business object available? Also when creating the runtime (configuration library) for the synchronization object, a number of things are checked. For example: are the specified attributes from the section defined as public attributes for the business object?

Before starting the actual event publishing server, a number of checks are executed to verify whether it runs properly. This is done to give direct response to the user (either through the user interface or through the API). An error in the runtime process is not detected directly by the user, so it should be avoided as much as possible.

These checks are done:

- Can the log file be written?
- Is the configuration library available and correct (for example, the library version and the definition of the public components and attributes on the one hand and the required tables and columns on the other hand)?
- Is audit activated for the require table(s)?

  Note

  The same checks are done when doing the corresponding actions from the Synchronization Server user interface, instead of using the *PublishChanges* method.

Transaction Management

The *PublishChanges* and *UnpublishChanges* methods execute database transactions when creating, changing or deleting the configuration settings. For example:

- After generating a synchronization object, the database transaction is committed before the runtime is created and the runtime process to publish the event messages is started.
- When stopping the event publishing a transaction is needed to inform the runtime process that it must stop. The communication is done through a database table.

Consequently, database changes may be applied even if the *PublishChanges* or *UnpublishChanges* method as a whole fail.

For example, when the starting fails, the synchronization object is already created. In that case, after solving the problem either a new *PublishChanges* can be done, or the existing synchronization object can be started from the user interface.

Error handling at runtime

While the server to publish change events is running, a logging mechanism is used. This means errors (which may either stop the server or not) are written to a log file that is linked to the server run. It can be displayed from the server runs sessions.

How to Solve Issues

If the *PublishChanges* or *UnpublishChanges* method fails, check the result as provided by that method and act accordingly.

If the method is successful, so a synchronization object is created, but the server for the synchronization object stops running unexpectedly:

1. Check the server log file. See “Logging and Tracing”.
2. If the log file does not provide enough information to solve the problem, use the trace facility get more detailed information.

If server runs correctly, but the event messages are incorrect or no event messages are sent at all you can:

1. Check the server log file. See “Logging and Tracing”.
2. Check whether the event you expect is correctly and completely written to the Audit Trail. If it is not there, either the update was not performed on the LN database, or the audit setup is not correct.
3. Check whether the selection and filter are correct. For example, are all required components and attributes listed? Are attributes set to *includeUnchanged*" as required? Is the filter correct? Are the constraints met as specified in “Selection and Filter”?
4. Check whether the *List* or *Show* method for the business object gives expected result when using the same selection and filter. Note that at runtime functions from the Business Object Layer are invoked to calculate the values of public attributes based on the values of the table fields. If correctly implemented, the attribute values for *List* or *Show* method should be in line with the attribute values as provided in an event message.
5. Use the trace facility to check whether the update is picked up from the audit trail and what is done with it.

In case of issues regarding performance or disk space usage, see “Performance and Disk Space Usage”.

Testing, Resetting, Rewinding

For testing, the *Test Synchronization Object* (danch2212m000) session is available.

At runtime, the Synchronization Server keeps track of the status for each synchronization object. This means the publishing can temporarily be stopped and continued at the right point in time later.

Functionality to reset a synchronization object is available in the *Synchronization Objects* (danch2510m000) session. Drawback is that all runtime data is removed, so the status is lost. The advantage is that the user can make a fresh start (at any point in time), for example if a problem is encountered and the server cannot get past this problem.

Additionally, you can ‘rewind’ a synchronization object. In other words, it can be stopped and started at a moment in the past.

This can be helpful if a client:

- Crashes and has a backup (for example from a week ago) installed. Then it can receive the changes from the last week again.
- Is unavailable temporarily and consequently change events are lost.

In these cases, you can rewind the event publishing using the Synchronization Server interface (simply stop the server and start it with a start time somewhere in the past).

This option has some limitations: (1) in case event messages from the synchronization object are distributed to more than one client application, *all* clients will get the ‘old’ event messages again, and (2) a performance issue occurs when unchanged data is included.

If the rewind cannot be used, the *List* or *PublishList* can be used to regenerate the complete data set for the client application. Note that in situation 1 this may even be cheaper than reloading the changes (depending on the backup age, data set size and change frequency). Additionally, situation 2 can be avoided by using a persistent message queue.

In theory you can run a synchronization object batch wise. For example, start it every day at 8 pm and stop after it completed processing the change events from that day. However, this approach will not be successful if any unchanged data is included in the event messages. For example, if *includeUnchanged* is set for one or more components or attributes in the selection, if unchanged data is included because of calculations in the business object layer, or if a filter is specified.

Logging and Tracing

Server Log

For each synchronization object, a log path is specified in the *Synchronization Objects* (danch2510m000) session. To view the log file for a server run:

1. Start the details session for the synchronization object
2. Choose the *History* command.
3. In the *Server Runs* (danch2520m000) session, select the server run and choose the *Show Log File* command.

This server log file tells you when the server was started and when it is stopped and for what reason. Additionally, it contains entries for generic errors that occurred and entries for objects that caused problems. Generic errors may cause the server to stop. In case of incorrect handling of specific objects, the server usually will continue processing the next objects.

The server log is in XML format. The structure of each log entry is:

```
logEntry
    DateTime
    Error
    Information
```

The 'Error' subtree contains generic information: error code, message code and message description. The 'Information' subtree contains specific information. It will always contain the server and run number. Additionally, it can contain information on the transaction and/or object that caused the problem. For example, transaction id and commit time of the transaction, and the image of the object at the moment the error occurred.

Examples of errors and warnings that can be logged:

- Cannot load configuration library (the specified library for the synchronization object is not generated or compiled or unavailable in the user's package combination).
- Cannot publish transaction data (the bus component may be unavailable).

This is an example of an entry from the log file:

```
<?xml version="1.0"?>
<LogEntry>
    <DateTime>
        <Date>03/08/14</Date>
        <Time>11:28:18</Time>
    </DateTime>
    <Error>
        <ErrorCode>-16</ErrorCode>
        <MessageCode>danchrun.16</MessageCode>
        <MessageDescription>Error running server 'reg002', run 1: creating net change failed</MessageDescription>
    </Error>
    <information>
        <server>reg002</server>
        <runNumber>1</runNumber>
        <errorLevel>Fatal</errorLevel>
        <errorPath>
            <errorLine
                file="pdanchxmltree0"
                line="3029"
                returnValue="-2"
            />
            <errorLine
                file="pdanchrunsteps0"
                line="1116"
                returnValue="-905"
            />
        </errorPath>
        <transactionId>10608532920000082607</transactionId>
        <commitTime>
            <Date>03/08/14</Date>
            <Time>11:28:12</Time>
        </commitTime>
        <errorLocation/>
```

```
    </information>
</LogEntry>
```

Trace

In addition to the server log file, a trace file can be created. The server log file is always available when a server is running. The trace file is an optional feature that is not created by default.

The trace gives more detailed information than the server log file. It shows the server activity, for example, what transactions were read and how they are processed. It logs not only exceptions, but also transactions that were processed successfully.

Caution:

Do not trace unnecessary, because this has a considerable negative impact on performance and consumes a lot of disk space. Tracing can be done at different levels:

- Low: The server activity is traced. For each process step an entry is created in the trace file. The actual business object data is not included; for each transaction, only the transaction ID and commit time is written.
- Medium: Same as Low, but the business object data from the transaction is logged for the first process step, just after reading from audit trail, and for the last process step, just before storing.
- High: Same as Low, but the transaction contents are logged for each process step.

The example shows the output of a low-level trace.

```
03/08/13 15:10:42 T_R010 READING          : No transactions currently
03/08/13 15:10:43 T_R020 READING          : Read transaction 10607802390000082607 having commit time 03/08/13 15:10:39
03/08/13 15:10:43 T_R040 READING          : Read action number 2 on table dareg101 having action type U
03/08/13 15:10:43 T_P020 PROCESSING       : One or more tuples to be processed for transaction 10607802390000082607
03/08/13 15:10:43 T_P030 PROCESSING       : Objects built for transaction 10607802390000082607
03/08/13 15:10:43 T_P030 PROCESSING       : Missing data read for transaction 10607802390000082607
03/08/13 15:10:43 T_B010 PROCESSING       : Transaction 10607802390000082607 enqueued
03/08/13 15:10:43 T_B020 PROCESSING       : Queue now contains 1 transactions (1 in memory), 1 objects (1 in memory)
03/08/13 15:10:43 T_R020 READING          : Read transaction 10607802390000344751 having commit time 03/08/13 15:10:39
03/08/13 15:10:43 T_R040 READING          : Read action number 2 on table dareg101 having action type U
03/08/13 15:10:43 T_R040 READING          : Read action number 4 on table dareg102 having action type U
03/08/13 15:10:43 T_R040 READING          : Read action number 5 on table dareg103 having action type I
03/08/13 15:10:43 T_R040 READING          : Read action number 8 on table dareg101 having action type U
03/08/13 15:10:43 T_P020 PROCESSING       : One or more tuples to be processed for transaction 10607802390000344751
03/08/13 15:10:43 T_P030 PROCESSING       : Objects built for transaction 10607802390000344751
03/08/13 15:10:43 T_P030 PROCESSING       : Missing data read for transaction 10607802390000344751
03/08/13 15:10:43 T_B010 PROCESSING       : Transaction 10607802390000344751 enqueued
03/08/13 15:10:43 T_B020 PROCESSING       : Queue now contains 5 transactions (5 in memory), 7 objects (7 in memory)
03/08/13 15:10:44 T_R010 READING          : No transactions currently
03/08/13 15:10:44 T_B080 PROCESSING       : Transaction 10607802390000082607 dequeued
03/08/13 15:10:45 T_B080 PROCESSING       : Transaction 10607802390000344751 dequeued
03/08/13 15:10:45 T_B080 PROCESSING       : Transaction  dequeued
03/08/13 15:10:45 T_B020 PROCESSING       : Queue now contains 0 transactions (0 in memory), 0 objects (0 in memory)
03/08/13 15:10:45 T_P040 PROCESSING       : Post-processing started for transaction 10607802390000082607
03/08/13 15:10:45 T_P050 PROCESSING       : filter tuple, library function 'filter.deliveries'
03/08/13 15:10:45 T_P050 PROCESSING       : filter tuple, library function 'filter.delivered.items'
03/08/13 15:10:45 T_S010 PUBLISHING       : Transaction 10607802390000082607 published
03/08/13 15:10:45 T_P040 PROCESSING       : Post-processing started for transaction 10607802390000344751
```

```
03/08/13 15:10:45 T_P050 PROCESSING     : filter tuple, library function 'filter.deliveries'
03/08/13 15:10:45 T_P050 PROCESSING     : filter tuple, library function 'filter.delivered.items'
03/08/13 15:10:45 T_S010 PUBLISHING     : Transaction 10607802390000344751 published
03/08/13 15:10:46 T_R010 READING        : No transactions currently
```

Debugging and Profiling

Usually a server runs in background in a Bshell without a user interface. This means a user can close the BW client after starting one or more servers.

However, for debugging or profiling a configuration library, the Synchronization Server must not run in background but in a Bshell having a user interface. This can be configured in the Synchronization Objects (danch2510m000) session.

Chapter 7 Publishing Development Guide

7

Development of Publishing Methods for Business Objects

For a business object, publishing methods can be added (*PublishList*, *PublishChanges* and *UnpublishChanges*). Since these are standard methods they must be taken into account when designing, creating and testing business objects.

Business object methods are linked to a business object in the *Business Objects* (ttadv7500m000) session. The publishing methods are comparable to other public methods, except that they have an additional argument specified. In addition to request, response and result arguments, also an *EventMessage* argument is linked to the method. Actually this is not a real method argument, but it specifies the format (XSD) of the messages being published asynchronously.

Development Considerations

Just like for any standard method it is important to take this into account:

- Delivering a public interface is a point of no return. Once a method is available or an attribute is available for a method, it cannot be removed in a subsequent solution or product version. It is better to deliver no method than to deliver an incorrect one. It is only possible to *add* optional attributes or methods to an already published interface.
- A public method requires a proper design, implementation and test. In the first place, not everything is possible for a *PublishChanges* method. Additionally, anything that is possible may not be desirable from a business, application or maintainability point of view.

Additionally, it is important to be aware that a publishing method is a generic method for a business object. It should *not* be written towards a specific integration domain. For example, what is logical for a CRM (customer relationship management) integration may not be logical for a production planning integration.

By default, the message definition (as stored in the XSD) should contain all public attributes that are persistent attributes for the business object itself. In some way or another these attributes are mapped to one or more fields of the business object's database tables. For example, for the sales order object, the *item* attribute should be included (because it is stored in a sales order line), but the

item description should not be (because it is stored in another business object, viz. Item). Processing attributes that are defined for specific methods of the business object and class attributes should not be included. For example, a *highestExistingOrderNumber* attribute is not a persistent attribute of an order business object.

If a solution tuned to a specific domain is required, it is possible to deliver specific synchronization objects. As said, do not misuse the standard publishing methods for this.

Overview

The development of publishing methods is comparable to the development of any business object method. So first the business object must be defined in the *Business Objects* (ttavd7500m000) session including its components, attributes, etc. Then the public methods *PublishList*, *PublishChanges* and *UnpublishChanges* can be added, including their method arguments. For each method argument an XSD (XML Schema Definition) must be created (or generated and adapted).

The only difference between the publishing methods and other methods is that the *PublishList* and *PublishChanges* methods have a fourth ‘argument’ called *EventMessage*. This argument actually is not an argument of the method, but a definition of the event message being published after the method is executed successfully.

Constraints on Components and Attributes

Note

In the Business Object Repository, a business object without a component can be defined. In that case, we regard the business object itself as the top-level component. Keep this in mind when checking the following constraints.

Component and Attribute names

In the event message of the *PublishChanges* method you *cannot*:

- Use an attribute name that is equal to a component name.
- Use the reserved word ‘PreviousValues’ as a component name or attribute name.

Optional Elements in the XSD

In event message of the *PublishChanges* method:

- All child components must be optional (in the XSD: minOccurs = 0).
- All attributes must be optional (in the XSD: minOccurs = 0) except the primary key attributes.

Constraints on Identifying Attributes

In the XSD for the event message of the *PublishChanges* method, for each component that is included *all* its identifying attributes must be included. For example, component Order has attribute orderNumber, and component OrderLine has attribute lineNumber.

Identifying attributes for a component must be mapped to primary key columns of the root table. It is not allowed to use other columns or data from other business objects.

Constraints on Attribute Types

In the event message of the *PublishChanges* method you *cannot*:

- Use a text attribute (unless you want to publish the text number). If an attribute is mapped one-to-one to a text table field, the content of the text is not included, and consequently (changes to) the text contents are not synchronized. Only the text number is sent to the client.
- Use an attribute having a complex data type (such as an xml structure).

Constraints on Component and Table Relations

In the XSD for the event message of the *PublishChanges* method you *cannot*:

- Use components where the component relations are not implemented through the primary key attributes, or where not all primary key attributes of the parent component are included in the relation.
- Use components having multiple tables where the table relations are not implemented through the primary key attributes, or where not all primary key attributes of the parent table are included in the relation.
- Include a component that uses the same table more than once, include two different components that use the same table, or include attributes from a component that map to a table having a 'Value' set in the Business Object Component Table Relations.

  Note:

  A table can be used only once in a synchronization object (and consequently only once in a component). Additionally, the 'value' that can be defined in Business Object Component Table Relations is not supported in synchronization methods. This holds for both root tables of components and non-root tables.

  For example, in the LN SalesOrder business object a 'value' is set on the root table for some components and the same table is used as a root table in multiple components. The Synchronization Server does not support this.

  Additionally:

- A used component must have at least one table. In other words, the component must be mapped to one or more tables in the database. Additionally, if a component is mapped to more than one table, the cardinality of the relation between those tables must be one-to-one.
- All non-root tables in a component are regarded as direct children of the non-root table. Note that the Business Object Repository doesn't imply this (in fact the relation between tables within

a component is not at all modelled in the Business Object Repository), and the *List* and *Show* also handle this differently, based on table references. However, in the end the result is the same, because the relations within a component are always one-to-one.

Constraints on Attribute Mapping

Only attributes that are mapped to a business object's persistent data must be used. Processing attributes that are defined for specific methods of the business object or processing attributes that are calculated from multiple object instances must not be included. For example, a *highestExistingOrderNumber* attribute (which is not related to a single persistent order instance but provided by a 'class method') cannot be synchronized.

Regarding calculated attributes please note:

Calculations as modelled in the Business Object Repository are used in the synchronization. However, the assumption is that the calculation is modelled completely. So if we ‘set’ the relevant protected (private) attributes that are mapped one-to-one on table columns, and we subsequently ‘get’ the public attribute that is mapped to those protected attributes, we will get the right value.

Consequently, incorrect values occur if 'work-arounds' using hand-crafted code are implemented, such as:

- Coding calculation logic in the wrong place in the Business Object Layer libraries.
- Using values from other attributes that are not included in the mapping definition for the public attribute.
- Using an attribute that is mapped to nothing, but in the implementation code selects data from the business object's table(s). For example, for an order line attribute, program an SQL query on the order header table to calculate the value, or invoke an existing library function that does something like that.

Public attributes can be calculated or derived from one or more table columns that are not in the root table from the component to contain the attribute. In that case, there must be:

- either a one-to-one relationship between the table (or tables) and component involved
- or one-to-many relation, provided that the resulting attribute is in the 'many' component (child), not in the 'one' component (parent).

  For example, a public attribute calculated from values from an order lines table cannot be included in the header component. But a public attribute derived from the header table can be included in the line component.

Constraints on Associated Objects

You *should not* include public attributes that map to persistent data that is not owned by the business object, such as a description from an associated business object. Technically this is not forbidden, but changes on instances of the associated object are not detected as changes on the object for which publishing is done.

In theory, data from other (associated) business objects can be added. For example, add the itemDescription to the OrderLine component. However, in case *PublishChanges* is used, this can only be done successfully if the value to be included is completely static.

The reason for this is that changes on the item description belong to another business object and will not be picked up by the Synchronization Server. Consequently, a client receiving changes on orders will not be notified is the item description is changed. If the client must be notified of changes in the item, a synchronization object for the item object has to be created (and consequently the item description is not required in the order object anymore).

When implementing a calculated attribute that uses data from tables outside the business object's scope but without modelling a business object relation, the constraints as mentioned earlier also hold.

Constraints on PublishList versus PublishChanges

The message for the *PublishList* and *PublishChanges* methods must match. More specifically, all components and attributes that are listed in the *PublishChanges* message XSD must also be in the *PublishList* message XSD. It is technically possible but not advised to have more attributes in the *PublishList* message.

Test and Delivery

Testing

First invoke the *PublishChanges* method for the business object. This creates a synchronization object for the business object. Then follow test procedure from “Testing and Deployment”. Finally, invoke *UnpublishChanges* or cleanup the synchronization object by deleting it from the *Synchronization Objects* (danch2510m000) session.

For testing *PublishList*, you can directly invoke the method and check the response.

Delivery

Delivery of the publishing methods and its XSDs is done automatically with the application package.

Developing Specific Synchronization Objects

Usually, synchronization objects are generated at runtime for specific customers, based on existing business objects having a *PublishChanges* method defined. However, for a specific domain also synchronization objects can be created and delivered.

For example, synchronization objects on business partners, quotations, sales orders and items for a sales integration, and synchronization objects on items, production orders, bills of materials, work centers, and routings for a production planning integration.

Such a specific synchronization object should not have the name of the business object (because then it might get overwritten by a generic synchronization object). It is advised to use both the business object name and the domain in the name for the synchronization object. For example, the Item business object can have a 'SalesItem' and a 'ProductionItem' synchronization object.

For synchronization objects that are defined in the context of a specific domain, the restrictions as defined for general synchronization objects are valid. But additionally, only components and attributes that are relevant for that domain must be included. For example, for integration with sales applications, the synchronization objects for sales order, item, etc. will have different attributes than those used to integrate planning applications.

Development Process

The development process consists of creating the synchronization object by selecting all components and attributes that are:

- Relevant for the outside world in the context of the domain and
- Meet the constraints as defined in “Manually Setting up a Synchronization Object” and “Development of Publishing Methods for Business Objects”.

Depending on the domain, a filter may have to be defined.

The development and test for a synchronization object are explained in “Manually Setting up a Synchronization Object”.

Delivery

You can deliver the synchronization objects with an application package or integration pack.

Based on the synchronization object, a configuration library (DLL) is generated. Each synchronization object has a configuration library defined (session danch2510m000). This library contains a number of predefined functions. The configuration library does not be delivered with the LN package, because it is generated at the customer’s site (using the create runtime command, which also generates the audit profile).

To deliver synchronization objects:

- Use the Export Synchronization Server Configurations (danch5201m000) session to export the configurations to a file.
- For this file, create an 'additional file' in the package the business object belongs to. This is done using the *Import Additional Files* (ttadv2270m000) session, which can be started from the *Additional Files* (ttadv2570m000) session.

A customer can use a delivered configuration through the *Import Delivered Configurations* (danch5204m000) session.