SAP
Internal

DMT Workgroup
Integration

Communication Paradigm

Version 1.1
4. April 2003

Authors:

Martin Deggelmann, Axel Hein, Joachim Kenntner, Frank Michael Kraft,
Günter Pecht-Seibert, Wolfgang Röder, Andreas Wildhagen

History:

4.4.2003 Version 1.1 following review by DMT WG Integration and PIC

24.3.2003 Review of draft version 1.1 by DMT WG Integration and PIC

7.3.2003 Informal review of draft version 1.0 by DMT WG Integration.

05.2.2003 Informal review of initial version 0.9 (FMK) by DMT WG Integration.

14.1.2003 Informal review of initial version 0.8 (FMK) by DMT WG Integration.

Contents:

1 Introduction

This paper aims to describe rules, which enable software components to run on a system landscape with multiple databases. Unlike a model in which all components in a system are run by one database, when software components are distributed throughout a system landscape, you must take into account a multitude of new rules. These rules are based on the following assumptions:

The business processes are divided into process steps that can be executed in different components. A process step (corresponds to a LUW) runs on a component. Components contain a set of business functions, which enable them to model these process steps independently. They are not dependent on the operational readiness of other components for this. The process is processed further by the component sending a message to another component, which in turn takes on the process. The message is transferred asynchronously.

A sender component cannot assume that the services of other components are available synchronously. It cannot have a status in other components and therefore cannot set any locks. It must guarantee that the process steps have been fully dealt with and that messages that it sends to other components have been sufficiently described.

A receiver component must be able to process these messages. It cannot assume that a sender will process the messages again and therefore solve its problems.

These assumptions will require some restructuring with regard to the way processes are handled in the system. In existing applications this would at most require a reimplementation of sub processes to adjust the application appropriately. This can be achieved with a simple encapsulation arrangement.

In doing so, the processes must remain simple otherwise the openness of the solution is lost and individual components cannot be replaced by external components. There are few exceptions to the assumptions above. However, one of them is the availability check with reservation. These exceptions are in part realized differently to how they are described here. However, this does mean that you should otherwise deviate from the rules described here. Every deviation from the rules above has to be approved.

2 Basic Rules

2.1 Prerequisites

2.1.1 Loosely Coupled Components

Components are units that can perform a business (sub) task independently. They have a set of methods for this task and have their own independent data (transaction data) that is generated in the course of process steps and that can be changed.

To enable loose coupling, the components must be arranged in such a way that a process step can be performed with having to communicate with other components. Communication with other components normally doesn’t take place until the end of a process step and is performed by an asynchronous message. Access to the services of external components is only permitted if they are configured so that a process step can still be performed even if the external component is not available (with restricted functions, if applicable). In this way, a component is defined by all the modules that are used to model the relevant process step.

It is not possible to carry out all verifications in all involved components in a scenario before a process step is executed. Each component is responsible for the completeness and accuracy of its data. It can complete a process step itself and send a message to the component involved. The business transaction must be completely described in this case. The receiver is responsible for further processing of the message.

If the components are divided up too much, then this will inevitably lead to restrictions in the provision of functions or a loss in operational security. The advantages of a division of components (more flexibility, for example) need to weighed up again these disadvantages.

If processes run across various components this may lead to conflicts across system boundaries where changes have been made. The best way to avoid this problem is to ensure at the design phase that an object only exists in one system or can only be changed in one system. Nevertheless, conflict situations will

arise as a result of the varying dependences of the objects in the process (for example, between a purchase requisition and an order). These can only be avoided if you use a pessimistic process flow model or they can be permitted by an optimistic process flow model. The pessimistic model stops conflicting changes by requesting a confirmation after a message has been sent, for example. However, this requires tight coupling. The optimistic process flow model on the other hand enables conflict situations. Once a message has been sent, the sender continues processing and assumes that in a normal situation no negative response will arrive from the receiver. This enables loose coupling, however you still require a strategy for solving conflicting changes, known as forward recovery.

2.1.2 Openness

In terms of a business process, a component is open if:

- The communication technology used is international standards (XML, SOAP, and so on), instead of proprietary communication means such as RFC.
- The interfaces are published in a central interface directory (such as the XI Integration Repository).
- The interface signature can be mapped to open standards (RosettaNet, UBL/ebXML, and so on).
- As little integration knowledge (such as receiver determinations, process choreography, and so on) as possible is contained in the components themselves.
- Process flow models are used for B2B processes (business to business) which can be mapped to open standards (RosettaNet, ebXML BPSS, and so on).

A component that is open in this sense can be replaced by a similar component with little effort. In particular, note that no enhancement is required in the other components, which are involved in the business process.

With SAP Exchange Infrastructure (XI), SAP has created a new basis for the communication of components.

- With the introduction of XI, SAP is using international standards, which widen its technological base. XI is available to all application components and enables SAP and non-SAP components to connect to each other.
- The communication between applications and between applications and business partners is no longer configured in the applications. Instead, the XI Integration Content is stored at a central location and describes important aspects of the subsequent communication. This includes receiver determinations, mappings, and interface descriptions.

The Integration Directory is the central location for the definition and storage of integration configurations. As XI becomes more and more widespread, SAP will be able to bring uniformity into the communication between components. However, openness is not guaranteed as a result. This must continue to be guaranteed by the application developers. Therefore, using XI will not automatically solve your semantic integration problem, rather it will help you to control it.
To ensure semantic integration and openness, you require a series of additional measures. Some of these measures are discussed in other papers (Interface Paradigm, Data Types) The aspect of openness is particularly relevant in this paper for the process flow model (choreography) and the communication technology.

The aim of implementing loosely couple components is to use optimistic process flow models (see section 2.1.1 Loosely Coupled Components). Generally, these are more demanding than pessimistic models, since additional communication steps may be required to solves conflicts. Certain open standards (such as RosettaNet) use pessimistic process flow models, which lead to a tighter coupling of components. To solve this target conflict, it may be necessary in SAP applications to implement another model in addition to the functional and richer optimistic process flow model. This additional model would only permit restrictive changes but can be mapped to open standards.

2.2 Basis Rules for Loose Coupling

There are several basis rules that need to be followed to reach the aim of loose coupling. These basic rules have a big influence on the design of components and their interfaces, and the interaction with their environment. These basic rules are explained in this chapter and are summarized again at the end.

2.2.1 Component Independency

Between the components in a system landscape there should be as few dependencies as possible as far as availability is concerned. The components must only be loosely coupled, otherwise the availability of individual components or systems is multiplied, which leads to a correspondingly bad overall availability. A component must be able to perform a business step without being dependent on another component.

You must ensure that no validation is carried out in another component. All the verifications that are performed locally must be sufficient to provide the other component with a complete result. In this case it is permissible to have optional communication with another component to enhance the range of functions provided. However, if the other component is not available, the process step must be executable (for example, in the status 'unchecked'). The result of the process step must still make sense and it must be saved consistently, or if applicable, a rework must be provided.

Further checks take place in the next component. This checks must not stop the process from running (see section 2.24). For this reason, it is essential that a message describes a business transaction with sufficient accuracy and that it sends the relevant attributes. The openness increases as a result, because the assumptions about another component are reduced.

2.2.2 Asynchronous Communication

At the end of a process step, each component must provide the result asynchronously for the further processing of the overall process. The process is always in the foreground. The result is passed on as a message by means of the Exchange Infrastructure. As a result, a significant amount of openness and flexibility is gained. For example, a message can be passed onto a system that is unknown at the time of development. This can be an SAP system or a third-party system. This makes it possible to envisage solutions where it is not necessary to make a fundamental distinction between intracompany and intercompany business processes.

In principle, because asynchronous communication increases availability, it is preferred to synchronous communication. However, there are some exceptions that can be better realized with synchronous communication. These are explained further in section 2.2.9.

Simply using asynchronous message alone does not guarantee a loose coupling of the systems. If an application is programmed so that a query to another system is performed asynchronously, but it still waits for the response to arrive, the applications are nevertheless tightly coupled. Applications are only loosely coupled if the user can continue to work before the response has arrived. There are appropriate mechanisms here (for example, which set the status to "in transition") that ensure that only changes are made to the objects concerned, which will not be invalidated if a negative response arrives back at the sender application (see sections 2.2.3 and 2.2.5).

With asynchronous communication and component-independence there is a paradigm change, similar to R/3. While a pessimistic approach was chosen in R/3 to keep data consistent in all involved applications and to avoid errors in advance (early locks, updating the data in another application in a LUW, checking all the subsequent components involved before updating the data), this is not longer possible with asynchronous communication. A component performs its process step according to its rules (though checks to data in other components could be carried out) and saves the results. The results of this process step can lead to conflicts in subsequent components when it is processed further (for example, because a subsequent action has already taken place). However, a rollback in the sense of a LUW is not possible here. Situations such as these must be taken into account by the applications in future, and appropriate mechanisms must be make available to solve such conflict situations (for example, by using Alertmonitor, reworks, mechanisms for forward recovery).

Hence forth, application development needs to be aware that the transaction concept in the sense of R/3 (ACID principle) will be replaced by the new communication paradigm. Results of process steps within an overall process will be made visible before the overall process is complete, and objects will be unprotected (in other words, unlocked) within an overall process. As a result, new concepts need to be developed to ensure the consistency of an overall process (forward recovery instead of rollbacks, alert mechanisms, logical locks such as status, and so on).

2.2.3 Solving Conflicts on a Business Level

Each problem must be solved on the level at which it occurs. If there is a business conflict, it can only be solved on a business level. There is no generic mechanism in the communication infrastructure that can solve this kind of conflict on a technical level.

The basic model for this is forward recovery in the process, as described in section 2.2.4. The following alternatives usually exist:

1. A local solution, which solves the conflict in such a way that a certain deviation between the systems has to be accepted.

   An example of this would be to transfer a receipt to the next accounting period if it arrives after the period has ended.

2. A more extensive solution because this deviation would not be justifiable. This could result in further process steps in various systems.
   Example: A change to a sales order position in CRM via FC leads to a request to SCE to change the delivery. The SCP is also informed about the change so as to update planning. However, since the delivery concerned has already been made, SCE refuses the change by means of a message to FC. The sales order is updated in CRM, and, for example, a returns document is created and planning in SCP is also updated.

What priority each method of forward recovery has, can only be determined in the context of the business requirement and the requirements of the particular scenario.

Conflict situations often arise because objects can be changed redundantly in various systems. Where possible, this should always be avoided by dividing the components appropriately. If objects are nevertheless distributed among different components so that they can perform different tasks (such as purchase requisitions in the planning system SCP and in the purchasing system SRM), then the overall consistency of the objects must be ensured. This can only be achieved if the change autonomy for this object is always clearly settled. This can be ensured by the following rules:

- Clearly dividing the responsibility of attributes of a distributed object (data-disjunctive responsibility) and the lock detail connected to it.

  Example: Amount and date for a purchase requisition are managed in SCP and communicated to SRM, while confirmed amounts and dates are managed in SRM and communicated to SCP.

- A chronological division of the responsibility of a distributed object (chronologically-disjunctive responsibility).

  Example: Once the purchase requisition has been transferred from the SCP to the SRM, it can no longer be changed in the SCP. The change responsibility therefore switches with the transfer to SRM.

2.2.4 Forward Recovery of Conflict Situations

Since sub results of an overall process are already persisted in components when using asynchronous communication, other mechanisms apart from a ‘LUW rollback’ or ‘lock all sub objects’ need to be used to solve conflict situations in an overall process. The receiving component must provide error handling if a fully described business transaction cannot be processed further due to insufficient settings. Similarly, the problem of postings to locked objects [1] or to periods that are already closed must be solved by the receiver component itself. Since the persisted part results can remain visible (unlike in a rollback), you refer to a forward recovery.

This approach also corresponds more to a real situation in the business world. Changes are carried out differently depending on the department and are then communicated to subsequent departments. If conflict situations occur in the subsequent department, then these are resolved by this department (including possible dealings with the department that triggered the conflict).

Essentially, it is possible to distinguish between different conflict and error situations and the way that they are handled:

[1] This a reference to configuration or application status locks, and not temporary locks because an object is being worked on.

- Business conflicts are not error situations. They arise because of conflicting change options in the process. These conflicts must be resolved by a forward recovery. This may be a local forward recovery program in the receiver system, but may also be a forward recovery log, that goes forward in the process and as a result starts new activities in other systems. You can also see a forward recovery as a reverse posting in the original system or returns, for example. Since process steps can be involved in multiple systems, these logs must be included in the design of the process right from the start.
- Errors that arise because there was a deviation from the organizational sequence in which the systems are run, for example when configuration data is not available in the target system. This error must be solved by a forward recovery in the receiver system, for example, by creating the missing configuration data in the target system and then processing the message again.
- Temporary locks. Such conflict situations are also solved by a forward recovery in the receiver system, by repeated attempts to process the message. This task should be performed by a generic framework. In the case of persistent locks, it will be necessary to make manual changes, as was the case in the previous examples.
- The aim of asynchronous communication and loose coupling is that the downtimes of individual systems should not lead to error situations occurring.
- Error situations caused by deleted messages, program errors, or technical communication errors. This also includes the case of the point-in-time recovery, in other words, when a system was forced to make a database recovery. This system is then not as up-to-date as the other systems. In the design phase it was possible to minimize these error situations by using appropriate measures, however they cannot be excluded completely. These error situations can only be solved by correction reports or by human intervention. This requires good monitoring tools, see section 2.2.7.

A forward recovery is not a rollback. The following are the most important differences:

- Rollbacks are performed at database level, while forward recoveries are performed at application level.
- Following a rollback, the final status is the same as the initial status. However, there may be differences following a forward recovery.
- In a rollback, different parallel processes have dependencies. All objects involved are locked and are returned to the initial status together. A forward recovery does not involve any locks, and a process can be ‘forward recovered’ independently of any parallel processes.

2.2.5 Data Consistency

Asynchronous communication can always lead to temporary imbalances occurring between the components. You should also not attempt to avoid this by using complex mechanisms (see also section 2.2.8) Such imbalances can last longer than the pure asynchronous message transfer. For example, a workflow can be started and the duration of the resulting imbalance cannot be determined.

Therefore, when programming, you cannot assume that the situation in a component is not up-to-date. Nevertheless, cross-component processes must be able to function regardless. Usually, this also corresponds to the connection between the system and the real world. There are always certain deviations between the real world and the current status in the system (for example, warehouse stock, unplanned deliveries). The situations that arise as a result are similar to those that arise if components are coupled asynchronously. Normally it is not possible to differentiate between a real inconsistency and a temporary imbalance just by using the data in the system. For this reason, methods for avoiding real inconsistencies in advance are difficult to implement.

Nevertheless, some imbalances are not admissible. For example, the must be no imbalances between a debit posting and a credit posting. Therefore, this posting must always be performed in a LUW in a component. This is only possible if the posting is sent to the component with one single message. If two messages are sent to a component, these will always be posted as two LUWs by the receiver component (even if messages are to be bundled for performance purposes, individual messages are always handled as individual LUWs).

The cross-component consistency can then only be guaranteed once all messages in a process have been successfully processed. If, for revision purposes, you were interested in completed business processes (for example, from the previous period), this would be the case. For current data in a productive environment, this would represent more the exception than the rule. For this reason, mechanisms and tools need to be

designed to monitor the consistency of data across all components, and to reestablish it, where necessary. In this case you also need to take into account that individual components may not be up-to-date as a result of database problems (point-in-time recovery). Consistency mechanisms must take this situation into account and enable the components concerned to be restarted in the overall context.

The tools for identifying and resolving inconsistencies between systems should follow the same pattern as described below: They should take into account that temporary imbalances can occur. It is not possible to include all messages that are still in transit at a particular point in time in the comparison. Instead, imbalances should initially be indicated on the screen and a refresh function should be provided in the monitor. If certain imbalances disappear when you refresh the monitor, then these were obviously just temporary differences. Only those differences that remain once the monitor has been refreshed numerous times should be analyzed in more detail. This relatively simple to implement approach has so far proved successful in APO.

When striving for internal and external component consistency, the two often end up in a conflict of aims. By agreeing that conflicts are always handled in the subsequent system, in the case of a conflict, messages in the subsequent system are not handled directly, but they may require a part rework. If messages now start building on each other, for example, on changes made to the same object, then the application is faced with the question of whether subsequent messages require the successful processing of previous messages are not (this in turn has an effect on the design of the messages with regard to current state or delta transfer).

If the subsequent messages require the successful processing of previous messages (and therefore insist on serialization, see section 2.2.7), then if a conflict arises, all subsequent messages must also be subjected to a rework. In this way, the internal component consistency is maintained, however the external component consistency is put more and more at risk. Even more complicated is the problem of conflicts in combined messages (multiple objects to receive the LUW in a message). This can quickly lead to uncontrolled dependencies (serialization effect), which in this procedure can in turn lead to extreme cross-component inconsistencies.

If subsequent messages do not require the successful processing of previous messages (and therefore do not require serialization), then combined messages may possibly contain internal component inconsistencies. The following example explains the problem in more detail:
Goods received for delivery are entered in SCE. This message cannot be processed in the subsequent planning component SCP and is transferred for reworking. The delivery is then changed in SCE (for example, confirmed amount). This message can be processed in SCP. If you were to transfer the change as a 'Current State' message and to copy all the attributes in SCP from the message to the delivery object (including the delivered amount), then you would create an internal component inconsistency. This is because the delivered amount in the delivery has already been updated, but the increase in stock has not. In subsequent processes this would lead to stock being ordered unnecessarily. To solve this problem, you could either just work with delta transfers or design the change delivery screen so that the delivered amount cannot be changed.

2.2.6 Using Patterns

The complexity of communication in a loosely-coupled component landscape requires a cross-component, common procedure for designing components and cross-component processes. If you compare different processes with each other, you will be able to identify many similarities and then be able categorize these processes.

When you categorize these processes, you will be able to identify and define underlying process patterns that satisfy the requirements of a process and that include the aspects of the loosely-coupled component landscape in the best possible way. There are two kinds of patterns, basic patterns and combined patterns. Furthermore, there are patterns that match the default paradigms, which are permitted, and others that do not match the paradigms, which are not permitted. Ultimately, the patterns that are permitted come from the rules of the paradigms. The permitted basic patterns are explained in chapter 12.

Of course, no set of rules exists that is so accurate that it can cover all possible scenarios. Therefore, it is essential that the pattern and the application of the pattern interact with each other. Nevertheless, if a pattern is available then it should always be used. Furthermore, you should always use the simplest pattern that solves the problem. If no appropriate pattern exists for the problem in question, then this needs to be discussed in detail, and, where applicable, a new pattern may have to be developed.

2.2.7 Dependencies Between Messages and Serialization

Within cross-component processes there are often dependencies between messages. The following are examples of such dependencies:

- Dependencies between Create, Change, and Delete messages from a sender. If these messages arrive at the receiver in a different sequence then this can possibly lead to problems. This section contains more information about possible approaches as to how to solve this problem, such as the serialization of messages.
- Dependencies between the request and response message. The hiding of the response message cannot be solved on a technical level rather it must be taken into account by the application. This is explained further in section 3.2.3.
- Dependencies between messages with three or more involved parties. This cannot be solved by a technical serialization (see below). A serialization by means of a request/response also has its limits. This problem cannot be overcome without intelligent application design.

The dependencies between messages with two involved parties can be represented in different ways.

- Technical serialization by means of queues. The communication infrastructure, for example, XI, ensures that messages arrive in exactly the same order as they were sent.
- Serialization by means of a request/response. The sender application always waits for the response from the receiver before sending the next message.
- Tolerant programming at application level. For example, old messages can be ignored if more recent messages have already been processed in the meantime. However, complete information about the object concerned must always be sent in this case. Furthermore, a message must contain at least one version so that the receiver can recognize that it is out-of-date.

  In some cases, it is possible to send difference messages instead of sum messages, for example if just figures (“amount ordered”) are to be transferred. In these instance, a message that has been overtaken only leads to a temporary incorrect status in the target system. As soon as the message, which has been overtaken arrives, then the status is correct again. However, messages must not be lost as a result of this procedure, otherwise the incorrect status will become permanent.

The technical serialization by using queues is only recommended and permitted if there are only two communication partners, and if one of the partners is responsible for changes to the object concerned. All changes must be applied by the other partner. No business conflict situations must arise as a result.

Also note that when using technical serialization there is the disadvantage that queues can become blocked if an error occurs.

Due to the problems described in section 2.2.5 you should avoid serializing messages where possible (regardless of whether you use a queue or request/response). However, if you do not use serialization, you must pay more attention to the design of the distributed objects, the interfaces, and the change autonomy, to ensure that cross-component consistency and internal component consistency is maintained.

2.2.8 No Two Phase Commit

A Two Phase Commit Protocol[2] should enable you to write data consistently to two systems. This can occur either at technology level or at application level, however the former is not possible with SAP Web AS technology. A Two Phase Commit Protocol should also not be used at application level in a loosely coupled component landscape. This is due to the following:

- The processes are extremely dependent on the availability and correct specification of all components.

[2] The Two Phase Commit Protocol should provide consistency in distributed systems. A commit is divided into two phases, the first being the provisional commit and the second the final commit. Two variants exist for the former: Either the provisional commit is visible to other transactions, or it is not visible to other transactions. Generally the second variant is preferred since the first variant requires cascading rollbacks. In the second phase it is assumed that no errors can occur.

- The probability that an error will occur at application level in phase 2 is too great. The checks that were performed in phase 1 are already out-of-date when phase 2 is reached.
- No open standard exists for the implementation of the Two Phase Commit Protocol.

2.2.9 Synchronous Remote Services

The basic rules refer to decoupled processes with replication and asynchronous communication. However, in a distributed component landscape there will also be processes that cannot be realized with these basic rules or that can only be partly realized.

In particular, this includes competing write services that are used by various different components.
Example: An ATP check for a particular material can be performed by CRM in the creation of a sales order, by SCE when goods go out for delivery, and by production when executing production processes. Since these processes are all competing for the availability of the material, a replication excludes the availability data in the connected components because the situation can only be recognized centrally.

Such processes can only be realized properly by using remote services. The question then arises whether these services are performed synchronously when performing a process step or asynchronously after a process step has been performed. In the synchronous case note that the unavailability of the service cannot lead to the process step also being made unavailable and that no inappropriate reaction time ensues with regard to the availability of the external component. In particular mass data accesses must not be performed by using such services. In the synchronous case you must use timeout mechanisms and ensure that further processing of the process step is restricted. If the remote service is to be realized asynchronously, note that basis of the service request may change before the result has arrived which means that the result is useless. You should consider whether the triggering object should be protected against changes until the result has arrived (which would imply a return to tight coupling), or whether the result is simply ignored and just leads to a temporary change and repetition of the remote services. The need to decouple the components tends towards the second alternative or you could realize a synchronous remote service with the corresponding timeout mechanisms.

An additional category of (possible) remote services does not represent write check processes. If these are to be realized locally, then besides the replication of information you also need to implement local logic (the actual check logic) on this information. If information and logic from various components is required for the check, then you must implement various logics in the components performing the checks, depending on the number of components involved. If new components are integrated in the overall system landscape and they each need to contribute to a check, then the components performing the check must be enhanced with the new check logic. Furthermore, existing data models may also need to be enhanced to be able to locally persist the information for the new check logic. However, this contradicts the aim of increasing openness and flexibility. Here too you should investigate whether checks which multiple components must contribute to could not be made available as remote services so as to be able to integrate new check requests more flexibly.
Example: Good receipt for a delivery schedule in SCE requires a check on delivery tolerances (information from SRM), on open amounts in the delivery schedule in SCP, and possibly a calculation check in FI. An example of a possible enhancement to the check may be whether goods receipt requires a quality check or not.

Result: Remote services should only be applied in certain instances due to component independence and the runtime required. They should only be used if the process step to be performed can still be processed properly despite external components not being available and other restrictions. Check processes in which multiple different systems compete to write access a single object must be realized as remote services (for example, an ATP check). These can be performed either synchronously or asynchronously. If the asynchronous option leads to considerable restrictions in usability, then the service should also be performed asynchronously. However, the following principle applies here: the process step must not have to stop because the service provider is not available. When the service is synchronous then you must use timeouts so as not to block the triggering process for long periods of time. Both asynchronous and synchronous services must be implemented as stateless.

2.2.10 Summary of the Basic Rules

Rules

[1] A transaction in a component must be able to run completely independently.

[2] Communication of the results to further components takes place asynchronously.

[3] Business conflict situations are resolved at business level.

[4] Forward recovery should always be applied.

[5] Note that imbalances can occur when making asynchronous postings between systems. Data that is to be written to a system consistently must be contained in one message (corresponds to LUW).

[6] There is a limited number of permitted patterns that must be implemented in communication.

[7] If messages reference each other then you should avoid serializations where possible. There are particular prerequisites that you must adhere to if you want to implement technical serializations by means of queues.

[8] A Two Phase Commit Protocol is not available and must not be recreated by using the application.

[9] Synchronous remote services must only be called if the process step can still be performed even when the service is unavailable.

Conclusion

- A component must be divided in such a way that the process step required can run without communicating with other components (see 2.2.1).
- Synchronous communication must only be an optional part of an action that improves the quality of the action. If the called service is not available then there must be an alternative procedure without synchronous communication to enable the process to proceed (see 2.2.2 and 2.2.9).
- A component publishes the results of a process step for further processing in the next component 2.2.2).
- The result is published as a message that is processed asynchronously. Communication takes place by means of interfaces that are published in the Integration Repository of SAP Exchange Infrastructure. It is not possible to communicate by using IDOC, RFC, or qRFC (see 2.1.2 and 2.2.2).
- If changes are permitted in a process step that refer to the successful receipt of a message in another system then a response must be received to ensure that the message was processed successfully (serialization using request/response, see 2.2.7).
- No messages must be refused because of local changes. The sender cannot be held responsible for a local change. Delayed messages should be processed by local conflict handling where possible, but at least by forward recovery conflict handling at business level (see 2.2.3 and 2.2.4).
- Temporary imbalances between components (such as different stock levels) must be accepted in distributed component landscapes. Imbalances cannot be completely excluded since this would result in the application being too tightly coupled, for example by using a lock concept. The applications must have a certain level of business error-tolerance in order to be able to operate with admittedly short, but nevertheless possible imbalances (see 2.2.5).
- Within a process chain, a component that has transferred its result to another component for processing can no longer stop the process. This is only possible at business level, for example by using a reverse posting (see 2.2.3).

3 Elementary Patterns (Basic Pattern)

Patterns can be found in cross-component communication. Complex collaborations comprise the subsequent elementary patterns.

3.1 Uni-Cast / Fire and Forget

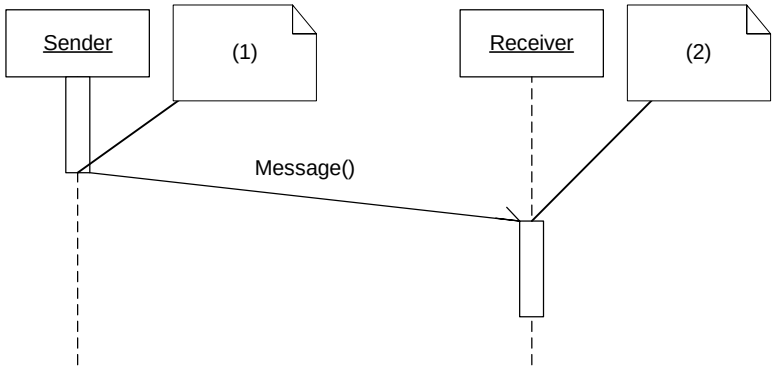
3.1.1 Definition

- Communication between exactly two partners: a sender and a receiver.

- The sender sends an asynchronous message to the receiver.
- The sender's responsibility for the message ends when it send the message.
- The receiver cannot refuse the message. In particular, the receiver cannot send back an error message.
- The sender can continue its process after it has send the message.

A uni-cast is just like sending a normal letter through the post.

3.1.2 Graphical Representation



(1) The sender sends an asynchronous message to the receiver. Once the message is sent its process step is complete.

(2) The receiver receives the delayed message and starts a subsequent process or performs an activity in the same process.

3.1.3 Comments

- The receiver is known to the sender in its *Role*.
- Error situations that arise because of undeliverable messages are not provided for at the sender. The error is corrected as part of a separate process.
- If a message cannot be processed at the receiver due to an error situation then it is up to the receiver to carry out appropriate measures for error handling.
- The sending of the message is a sender process step or it is an elementary part of a process step (a process step corresponds to a LUW).

3.1.4 Use

- The uni-cast/Fire-and-Forget pattern applies for application when a component has completed its tasks in a process and transfers the responsibility to another component (passing the baton principle).
- The uni-cast/Fire-and-Forget pattern is used if a new process is started within another process that then runs alongside and independently from the triggering process.

3.1.5 Examples

- The billing engine sends an invoice for an order to the relevant receiver. The role of the receiver is know. The receiver has been entered as the receiver of the invoice in the order. The company, which processes the order, is the sender. The business partner, which receives the invoice, is the receiver. Once the invoice has been created and sent the task of the billing engine is complete. If the invoice

does not reach the receiver by post or by electronic transfer, then a special error handling process is required.

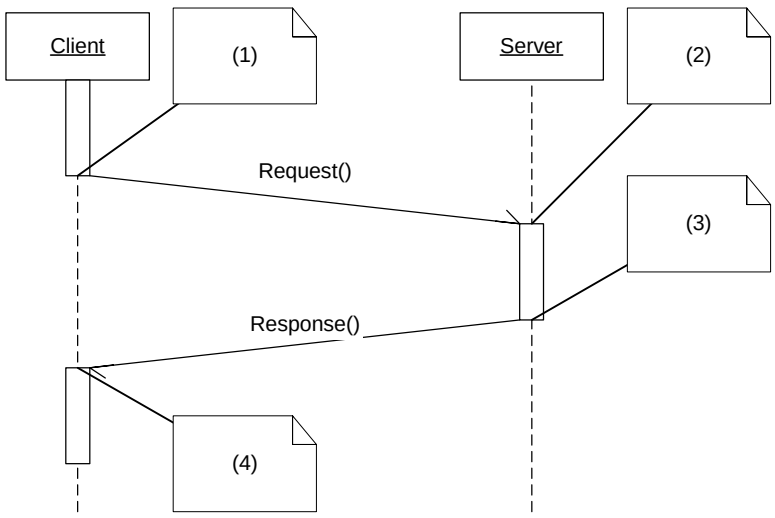
- The moment the invoices is sent, the billing engine transfers the request to the customer accounting department. The billing engine has thereby passed the baton onto the customer accounting department. The task of following up overdue invoices and so on is then the responsibility of the customer accounting department.

3.2 Request and Response

3.2.1 Definition

- Communication between exactly two partners: a client and a server.
- The client sends an asynchronous request message to the server. The server sends an asynchronous response message back to the client within a defined time period.
- The client waits for the response from the server before proceeding with the process based on the response (in reality the client can take on other tasks while waiting for the response here).

3.2.2 Graphical Representation



(1) The client sends an asynchronous request message to the receiver. Once the message is sent its process step is complete. The client waits until it has received the response before continuing with the next process step.

(2) The server receives the delayed request message and starts an activity or a sequence of activities (multiple steps with interruptions, if necessary) in the same process.

(3) The server sends the result of the activity/sequence of activities to the client in the form of a response message. This is the sender's final task in the process.

(4) The client receives the delayed response message. The process continues at the client.

3.2.3 Comments

- The server is known to the sender in its *Role*.

- The server does not need to know the role of the client.
- If the client does not receive a response within the defined time period, then it must begin an appropriate conflict resolution procedure. The shorter the defined period during which a response must be received, the greater the probability that a conflict will arise when the server has sent a response but it has not been received at the receiver in time. When the client then continues the process, the client and the server find themselves imbalanced. This status is recognized when the client receives the response and notices that the process has already been continued because the defined time period has been exceeded. In this case a process is required to resolve the imbalance problem. Alternatively, the delay to the response can also be made transparent at the client so that a workflow can be started to search for the cause of the error, for example. Which alternative you choose depends on the options available for continuing the process when the message is delayed in the application and how expensive it is to resolve any resulting imbalances.
- The sending of the request message is a process step on the client side or it is an elementary part of a process step.
- The act of processing the message at the server can trigger a process. The sending of the response message from the server to the client completes the server process.
- Receipt of the response message triggers the next process step at the client.
- Once it has sent the request message, the client process waits until it receives the response message or until the defined time period has been exceeded. The process is not blocked unlike in synchronous communication.

3.2.4 Use

- A component that drives a component (client) has another component (server) execute a process step or a sub process (service). Once the action is complete, it takes over control of the process again.

3.2.5 Examples

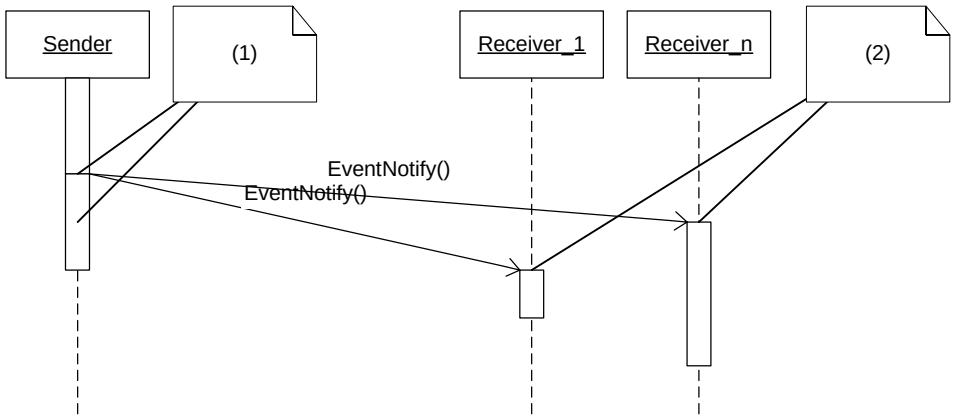
- A purchaser (client) sends an order to a vendor (server). The vendor must send either an acknowledgement or a refusal to the purchaser within one day of receiving the order. If the vendor sends an acknowledgement after this time period, the purchaser considers the order to have been refused and so sends the order to a different vendor instead. If the acknowledgement does not arrive within one day due to a communication error, then the purchaser and the vendor agree over the phone instead.

3.3 Publish and Subscribe

3.3.1 Definition

- Communication between a sender and any number of receivers (observers). In particular, it is not a prerequisite for the sender that there is at least one receiver.
- The sender does not block processes.
- The receiver does not send a receiver confirmation or a response to the sender.
- The sender does not have any means for handling errors in the case of problems during message transfer.

3.3.2 Graphical Representation



(1) A sender publishes an event in the form of an asynchronous message. Once the event has been published, the sender proceeds with its tasks independently of the receivers. The message can have any number of receivers (0..n).

(2) Each receiver receives a copy of the asynchronous message independently of the other receivers. Based on the message, the receiver either updates data or starts independent subsequent processes from the sender process. The receivers work concurrently.

3.3.3 Comments

- The sender publishes an event in the form of a message, such as reaching a certain status or changes made to data. It is not a prerequisite for the sender that on receipt of the message a particular subsequent action will be triggered at one of the receivers.
- One receiver observes and processes events. A receiver can also be seen as an observer.
- The sender does not know the roles of the various receivers. In particular, it is not a prerequisite for the sender that there is a receiver in a particular role since the sender cannot assume that that there are any receivers at all.
- The sending of the message is not optional for the sender. In other words, it is not a separate process step.

3.3.4 Use

- The Publish and Subscribe pattern is used when it is known that receivers exist but their existence is not a prerequisite. A development team from the sender application and representatives of the receiver applications determine how events are to be modeled. The pattern is only implemented if the existence of receivers is optional and not if the flow of the business process relies on the existence of receivers.

3.3.5 Examples

- The Publish and Subscribe pattern is used for the distribution of the master data changes made in the ALE scenario central master data maintenance to satellite systems. Once the initial data transfer (not part of this pattern) from the central master data system to a satellite system has taken place, the satellite system gets changes in the master data system because the ALE distribution model is maintained appropriately. If a change is made to a master data object in the central system, the changes are published as asynchronous messages to all satellite systems. The replica master data objects in the satellite systems are then updated accordingly. The existence of satellite systems is not a prerequisite in central master data maintenance. If problems arise when distributing the change messages, an explicit data comparison is performed by means of data extraction on the central system (not part of this pattern). Since a master data object can be changed many times in

succession, it is generally a prerequisite that the **sequence of changes is maintained** when the messages are processed at a receiver. See section 2.2.7.

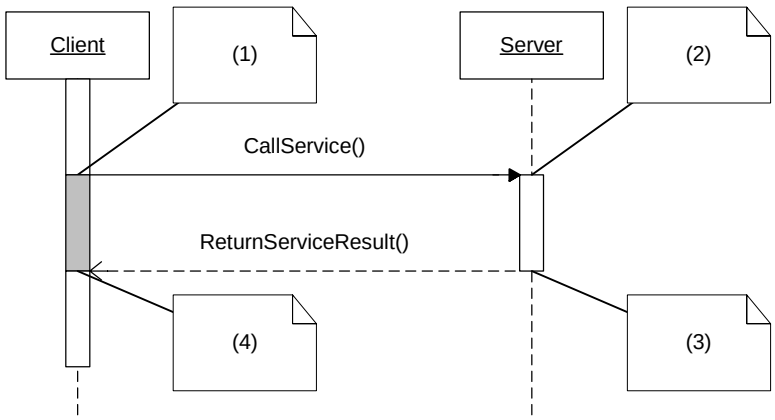
- The Business Warehouse collects reporting data for both master data and for reporting transaction data following the Publish and Subscribe pattern. For example, as a sender, the CRM procedure application can publish all changes to a CRM procedure. The SAP Business Warehouse does not play a role in the business process of the processing of the procedure. The business process runs both with and without SAP Business Warehouse (or another similar reporting component). Following a data extraction of all CRM procedures, all procedure changes are then published and send to the Business Warehouse as an event subscription. The sequence of changes is ensured by using queues.

3.4 Synchronous Communication

3.4.1 Definition

- Communication between exactly two partners: a client and a server.
- The client sends a request message to the server.
- The server processes the request message just like a function call, and creates a response message.
- The server sends the response message to the client.
- The client receives the response message in the same context (memory area,…) in which it sent the request message.

3.4.2 Graphical Representation



(1) The client sends a service request message to a server synchronously. Just like a sub program call, the client remains blocked until the result is received (4).

(2) The server receives the service request and performs an activity in one step (without interruptions).

(3) The server delivers the result of the service in a service result message. Once it has delivered the service response message, the server has no more tasks to carry out.

(4) The client receives the service response message. The client is now no longer blocked. The client continues its activity with the results of the service.

3.4.3 Comments

- Synchronous communication here involves one blocked call over system or component boundaries between exactly two partners: a client and a server. Synchronous communication therefore has large similarities with the asynchronous request/response pattern. There may be differing models in event-oriented program environments (for example, Microsoft Windows).
- The blocking of the caller ends on receipt of the response message or when an exceptional situation arises (communication error, timeout, and so on). The response message is available to the client following a successful call.
- If multiple synchronous communication steps are performed between a client and a server following the pattern above, and if the server thereby receives its context, then you refer to stateful communication. If the server contains no context between two synchronous communication steps, then you refer to stateless communication.

3.4.4 Use

- Synchronous communication should generally not be used for implementing distributed (cross-component) business processes because it restricts the availability of processes.
- Synchronous services can be used if they are either optional or can be used in high-available environments, and the communication is stateless. However, since every type of communication must be configured, security risks considered, and the failure risk increased, the number of synchronously called services must be kept to a minimum.
- Data updates are not permitted in synchronously called services. Even in certain justifiable exceptional cases, data updates for clients and servers in synchronous communication (→ 2 phase commit problem) can still be subject to problems.

The only known applications are availability checks for reservations and credit limit checks. If the server or the requested service is not available, then a client must be able to continue its process with reduced quality, if applicable.

Additional applications that communicate synchronously must be investigated further and must be checked and approved individually.

3.4.5 Examples

- Stateless synchronous communication is used in Web Services. The calculation of taxes for order items can be contacted by using stateless synchronous communication as a WebService. (OK)
- Stateless synchronous communication is used for read access (queries), for example in the F4 help BAPI. (OK with restrictions)
- Stateful synchronous communication is used between the SAP GUI and the SAP Web Application Server. (Not suitable for process integration)
- Stateful synchronous communication is used in the BAPI dialog program model. (Not suitable for process integration)
- Synchronous RFCs between two SAP Web Application Servers are stateful (programming can nevertheless be stateless). (Not suitable for process integration)
- Synchronous communication by means of SAP Exchange Infrastructure (XI 2.0) only supports stateless synchronous communication. (Not suitable for process integration)