(Inter)facing the Business

(Industry Paper)

Alexander Hagemann

Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: hagemann@hhla.de

Gerrit Krepinsky

Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: krepinsky@hhla.de

***Abstract*—Over the past decades, a change from singular mainframe applications into complex distributed application landscapes has occured. Consequently, the execution of business processes takes place in a distributed manner, requiring an extensive amount of communication between different applications. It becomes apparent that application interfaces are of overall significance within distributed application landscapes. But in our experience, interfaces usually do not get the required attention during construction, which is in contrast to their importance. Instead, only technical descriptions, e.g., syntactical descriptions, are given and important functional as well as operational aspects have been omitted leading to unstable and unneccesary complex interfaces. To address the aforementioned problems, this paper contributes a comprehensive overview on interface construction. Therefore, all necessary interface specification components, an interface design process and operational migration patterns are given.**

Keywords–interface; business process; interface design; interface migration; distributed systems.

I. Introduction

In recent years, growing business demands enforced an increasing information technology (IT) support of many business processes. To rule the resulting functional complexity within the IT, several applications are usually necessary. A direct consequence of this fragmentation is the distribution of business processes over applications which have to communicate with each other in order to fulfill the requirements of the business processes. This communication requires well defined interfaces between these applications.

Generally, the design of application interfaces is a difficult and critical task [1], [2], since the behavior of applications belonging to the class of reactive systems, i.e., applications responding continuously to the environment, is determined by their interfaces only [3]. Consequently, badly designed interfaces may lead to functional misbehavior and may propagate internal application problems directly to communication partners [4], [5]. Furthermore, interfaces have relatively long life-cycles and are usually costly to modify. A change of an interface specification always requires either its backward compatibility, or a change of all implementing applications, leading to further problems while launching the new interface into an already running application landscape [6].

Within the literature, a lot of information exists regarding different aspects of interfaces like performance, reliability, routing etc. Typically, these documents either deal with technical protocols only and omit functional interface properties, e.g., the internet protocol [7] or the Blink Protocol [8], or are bound to specific functional domains like the Financial Information eXchange [9] or the FIX Adapted for Streaming [10] protocols. But none of them gives explicit guidelines for an interface design. Other common approaches like service oriented architectures (SOA) [11] or the representational state transfer (REST) [12] represent rather general architectural styles. Both are more suitable giving architectural guidelines for application design, than for the construction of concrete interfaces.

To overcome the above mentioned problems, an approach to construct a consistent interface specification, allowing a regulated communication using stable, understandable and performing interfaces, and its transition into operation will be presented in this paper. Beginning with an overview of all required components to fully specify an interface in Section II, Section III introduces and compares different design approaches for the construction of interface specifications. Finally, Section IV deals with the interface launch into an already running application landscape.

II. Interface basics

In order to design an appropriate interface, the general structure of an interface must be considered. Once this has been done, it will become obvious which information must be provided to define an interface.

Drilling down into an interface, which is located in the application layer in the Open Systems Interconnection model (OSI model) [13], typically, a three layered structure, as shown in Figure 1, becomes visible. Each of these layers has a dedicated important purpose that can be summarized as follows:

- *functional layer*: this topmost layer is responsible for the functional semantics of the information exchanged. Using the analog of natural speech, the functional layer defines the meaning of words spoken.
- *protocol layer*: Within this layer the technical protocol used to exchange the information is defined. Similar to natural speech, the protocol layer represents the language spoken, e.g., English.
- *transport layer*: Here, the necessary physical transportation of the information is carried out. This layer

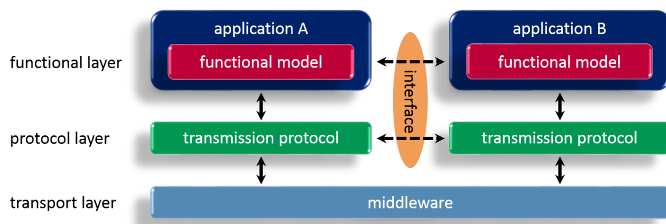

Figure 1. The different layers of an interface. Dotted arrows denote virtual connections within each layer. The communication takes part using the connections denoted by solid arrows.

correlates to the signal transfer using sound waves in a manner similar to natural speech.

Each of these layers communicates logically directly with its counterpart located at the other application. Therefore, a layer physically passes the information to its underlying layers until the information is physically transported to the other application. At this point, the information is passed upwards up to the corresponding layer. Only if both sides within one layer use identical functional models or transmission protocols, respectively, communication will take place. Otherwise, the communication is broken.

Given the layered structure of an interface, different aspects arise which must be considered during the design, implementation, integration and operational phases of an interface lifecycle. These aspects focus on different issues and enable the development of robust interfaces. All aspects are independent with respect to each other, focusing on a specific property an interface must satisfy.

A. Functional aspect

While two or more applications are communicating with each other over an interface, the applications assume different functional roles, called *server* and *client*, respectively.

An application is called *server* with respect to an interface if it is responsible for the business objects, business events and related business functions that are exposed to other applications through this interface. If business objects and business functions of different business processes are affected, the necessary access messages may be combined into a single interface.

Providing an interface is equivalent to defining an interface contract [6] that must be signed by an application in order to communicate with the *server*. The interface may support synchronous and asynchronous communication as well as message flows in both directions, i.e., sending and receiving messages.

Note that this definition deviates slightly from the commonly used client-server definition where the server offers a service which can be accessed by clients via a synchronous request-reply communication protocol only [14]. Because synchronous communications couples server and client tightly at runtime, asynchronously based communication should be preferred, avoiding these disadvantages [15].

A *client* is an application consuming an interface provided by a server. Despite the fact that the interface contract is initially defined by the *server*, a common agreement on the contract is made when the *client* connects to the server. Thereafter, none of the participating applications may change the interface contract without agreement of the other party.

Often an application assumes multiple roles with respect to different interfaces concurrently, i.e., the application can be *server* and *client* simultaneously. It is important to emphasize that this behavior is valid with respect to different interfaces only while for a single interface, the roles of the participating applications are always unambiguous.

B. Semantical aspect

The semantical aspect focuses on the kind of information that may be exposed by the *server* via an interface. Generally, any internal implementation detail of the *server*, i.e., the server model, must never be exposed on an interface. Instead, the information exposed must always be tied to the underlying business processes, thus binding the interface implementation to the domain model [6], [16].

Integration within an IT application landscape requires the decoupling of business and software design due to different responsibilities. In other words the business model and the software model usually have different life cycles which must be decoupled to reduce the dependencies between business and software developers. Therefore, an integration model, linked to the domain model, should be used on interfaces thus binding their implementation to the domain model [16]. The integration model finally conceals all internal application models and details.

An interface itself consists of a set of messages, containing *business objects* or *business events* only [17]. This set of exposed information is naturally restricted due to the responsibility of the *server*, i.e., only the *business objects* or *business events* the *server* is responsible for may be communicated via the interface.

C. Dynamical aspect

An important aspect of an interface is its dynamical behavior describing all valid message sequences on the interface. Since all messages received are processed within a specific context inside the application, there exist important constraints with respect to the message sequence. Thus, a message received out of sequence will not be processed by the application, instead this will result in an error.

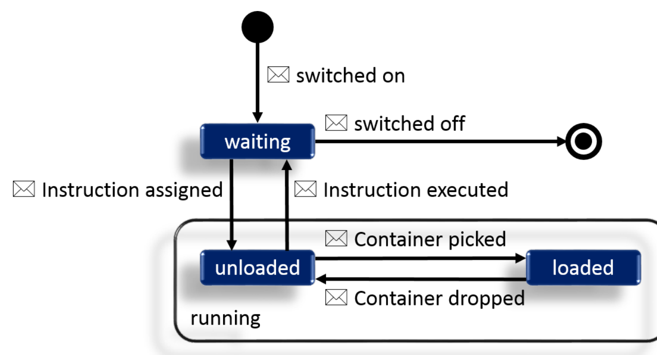

Figure 2. Example of a simplified state machine describing the dynamical behavior of an interface.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

Consequently, the dynamical behavior must be described using an appropriate description. Using sequence diagrams of the Unified Modelling Language (UML) is not sufficient for this case, since they describe specific communication examples only. Especially runtime problems, e.g., race conditions, can not be described holistically using sequence diagrams. Instead, it is strongly recommended to use finite state machines which allow a complete description of the dynamical behavior, see Figure 2 for an example.

D. Operational aspect

Usually, each interface requires the usage of specific infrastructure depending on the used transmission protocol, e.g., a web server in case of REST over Hypertext Transfer Protocol (HTTP) or a Java Messaging Service (JMS) server. To ensure the correct usage of an interface the required infrastructure, its deployment and the message channel topology must be defined. The latter one defines the communication structure, i.e., broadcast or point-to-point communication [15].

E. Interface components

Given the different aspects presented so far, each of them describing a different important issue with respect to interfaces, the necessary components for a complete interface specification can be derived:

- *message description*: Syntactical descriptions of all messages exchanged over the interface.
- *dynamic description*: The dynamic behavior of the interface must be fully specified. This specification includes all possible message sequences and the behavior of the applications in case of errors.
- *semantic description*: The meaning of messages on the interface must be specified, i.e., their functional behavior within the comprehensive business process. This description must include the meaning of all individual message fields.
- *infrastructure description*: A description of the necessary infrastructure must be provided.
- *quantity description*: The non-functional performance requirements for the interface must be described.

It is important to notice that an interface specification is a signed bilateral contract, which may be changed by mutual agreement of all participating parties only. This contract is represented by the set of artifacts as described above, so none of the artifacts given there may be missed.

III. Interface Design Styles

The main problem to be solved in interface design concerns the intended functional semantic on the interface. It directly influences the kind of service offered by the *server* and therefore the necessary number and style of all messages.

Looking at existing interfaces, they can be categorized to our experience by their semantic design styles: CRUD based interfaces, use case based interfaces and business process based interfaces, each of them described in detail in the following sections.

A. CRUD based design

The Create, Read, Update, Delete (CRUD) based design directly uses the business objects described within the requirements and ignores any given business context. This results in interfaces consisting of a minimal set of messages, representing a set of CRUD messages for every business object the *server* is functionally responsible for. Besides the advantages of requiring very little design efforts and being very stable, this interface design style has some important disadvantages.

First, the interface bears absolutely no business context, leading to severe difficulties in understanding the underlying business processes [1]. Second, the read operation demands synchronous communication which represents an explicit control flow leading to a tight coupling of applications [4], [17] and third, missing business context either leads to a distribution of business functions over the *clients* or to business objects incorporating the results of applied business functions.

B. Use case based design

An interface design based on use cases rests upon requirements formulated from the perspective of the primary actors for individual systems only [18], i.e., the underlying business process is not directly present. Due to the characteristics of use cases, describing non-interrupted interactions with the system [19] that represent the view of the primary actor [18], these requirements are limited to the context of single activities which are typically independent with respect to each other. A representation of the underlying business process triggering the desired activities is missing and therefore difficult to reconstruct.

These preconditions usually lead to rather fine granular and use case oriented interfaces comprising of a large number of messages, carrying specific use case based information only. Some important consequences arise from this design style. First, all business functions that are identical from the business process point of view, are hard to identify based on a use case analysis only. The absence of a business context leads to an interface design supporting individual use cases, which bear no evident business process semantics. Consequently, these interfaces usually offer a broad range of identical functionalities named differently. Second, the missing business context significantly increases the difficulty to understand the functional behavior of the interface over time [1], leading to serious problems in its usage. As a consequence, further unnecessary messages are often introduced in order to provide some use case specific information. Third, the lower level of abstraction of a use case - compared to the business process - leads to a rather fine granular interface structure. Performance issues may arise with this interface style due to the enforced frequent interface access [15]. And fourth, synchronous communication often arises in order to collect all necessary information to execute the use case, so a control flow arises [17] leading, again, to a tight coupling of applications [4].

Note, that using a use case based design must not lead compulsorily to a bad interface design. But given the size of current applications with their numerous use cases and the typical usage of distributed programming teams within industrial projects, the necessary refactoring to introduce an appropriate abstraction on the interface is usually omitted in our experience.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

C. Business process based design

This design uses business process descriptions and further requirements formulated with respect to those descriptions, to align between individual business process activities and applications. Using the Business Process Model and Notation (BPMN) and representing applications via pools, interfaces can be directly derived from the exchanged information between individual business activities within the pools.

The resulting interfaces focus on business semantics and directly support objects, events and functions of the business processes, thus leading to a business model directly bound to the interfaces [16] with following consequences.

Business processes support a high level of abstraction, thus leading to rather coarse granular interfaces with respect to the number of messages. The communication is driven by business events, so asynchronous communication is naturally supported, leading to data flows [17]. Finally, the functionality provided by the server within the business processes becomes rather clear, i.e., the business context is represented on the interface.

D. Design example

To explain and clarify the differences between these design styles, the simplified process of loading a truck at a container terminal will serve as an example throughout this section. This process consists of the following steps, executed in the given order:

- *order clearance*: the customer gives an order to the container terminal to load a container on a truck.
- *load clearance*: in order to deliver the container, several clearances must be given, e.g., by customs and the container owner.
- *transport planning*: the container terminal plans the necessary equipment to execute the order.
- *load container*: the container is loaded on the truck using the planned equipment.

Two applications shall be constructed in order to implement the process: the *Administration*, dealing with the administrative parts of the process, and *Operating*, handling the physical transport of the container. An interface between both applications will be designed according to the design style considered, thus showing the differences between the design approaches.

1) CRUD based design: All relevant business objects of the truck loading process are represented as classes which have methods to create, read, update and delete the object. These methods represent the interface of the owning application, i.e., the *server*, and are called by the *clients*, in order to execute the business process.

For example, after creating an order using `createOrder()`, the *Administration* calls `createInstruction()` to start the loading of the container on a truck. Subsequently, *Operating* calls `readCustomsClearance()` and `readReleaseOrder()` to check if the container is released to be loaded on a truck. The corresponding return objects must be interpreted within *Operating* to make this decision. If the container has been loaded, *Operating* finally calls `deleteOrder()`, `deleteCustomsClearing()` and `deleteReleaseOrder()` to clear the *Administration*.

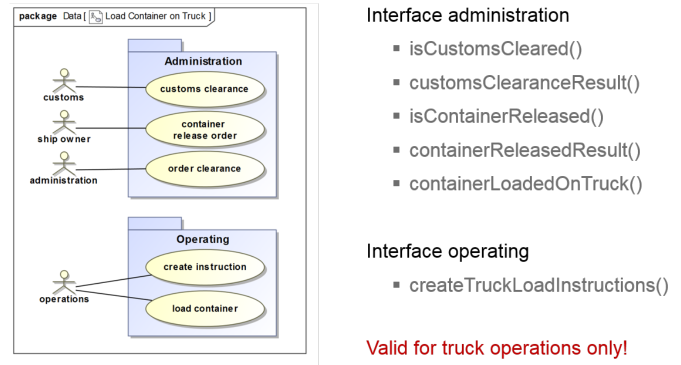

Figure 3. Constructed interface (right) resulting from applying the use case based design approach.

It becomes clear that both applications, i.e., *Administration* and *Operating* must implement some part of the underlying business logic to deal with these type of interfaces. Since the interface style bears no business semantics, the underlying business process cannot be reconstructed easily. Note that the size of the interface directly depends on the number of business objects the server is responsible for.

2) Use case based design: Based on the requirements of the truck loading process, corresponding use cases like order clearance or create instruction can be derived, as shown in Figure 3. Each of these use cases handles a specific functional aspect with respect to its primary actor. The underlying business process is executed through a set of use cases interacting with each other.

For example, if an order has been given, *Administration* calls `createTruckLoadInstruction()` to initiate the container transport. Prior to loading, *Operating* checks the container release status, using `isCustomsCleared()` and `isContainerReleased()`. If the container has been released, it is loaded on truck and *Operating* informs *Administration* via `containerLoadedOnTruck()` that the order has been executed. *Administration* may then clean up its internal data structures.

As depicted on the right side of Figure 3, the resulting interface contains a lot of methods for specific actions, i.e., the level of abstraction is rather low. Consequently the interface is valid for truck operations only and would require a couple of additional methods to incorporate e.g., vessel and train operations.

Furthermore the interface introduces synchronous communication, as indicated by, e.g., the method pairs `isCustomsCleared()` and `customsClearanceResult()`, leading to a blocking of *Operating* while accessing the information.

3) Business process based design: In this case, the business process itself serves as basis for interface design. Using BPMN, the process of truck loading can be mapped onto the applications as shown on the left side of Figure 4. Due to the given high level of abstraction within the business process, it is valid for all types of carriers, i.e., no further messages are necessary to include vessel and train operations.

Once an order has been given, *Administration* informs *Operating* via `orderPlaced()` that a new order has been

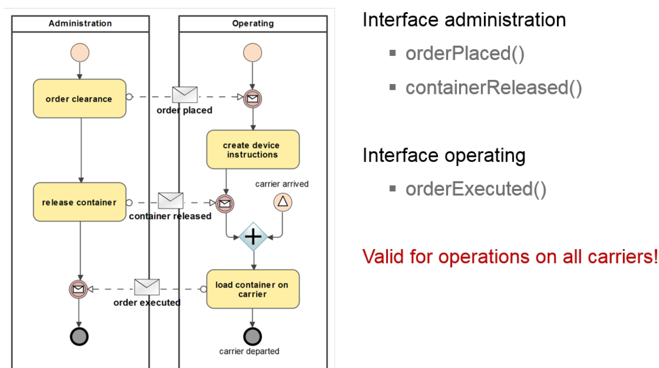

Figure 4. Constructed interface (right) resulting from applying the business process based design approach.

accepted. Within *Operating*, all necessary instructions for container loading will be created. Once the truck has arrived and *Administration* has published via `containerReleased()` that the container is released to be loaded on a truck, the physical moves are executed. Afterwards, `orderExecuted()` informs *Administration*, to clean up its internal data structures.

The dynamical behavior of the interface can be derived directly from the BPMN description, see left side of Figure 4. The resulting interface is quite small, meaningful and abstract, so other carriers can easily be included. Additionally, the communication between both applications is asynchronous. Note that both applications, *Administration* and *Operating*, do not technically depend on each other, instead they simply publish their information without knowing the receiver, resulting in a data flow [17].

E. Comparison

To give a recommendation for a specific interface design style all design approaches described above have been compared to each other using typical interface design goals like robustness, performance and understandability [1].

1) Robustness: Interfaces are crucial with respect to the stability of the overall application landscape. Poorly designed interfaces may propagate internal application errors during runtime, thus causing damage within other applications [4], [1]. Robustness is achieved by avoiding functional distribution, distributed transactions [5] and semantical ambiguity.

In case of a CRUD interface, the information provided by the interface must be functionally interpreted by the client since the *server* informs about changes on business objects only without any functional context. This leads to multiple and distributed implementations of business functions according to the usage of the interface. In contrast, the use case and business process based design styles can both concentrate the business functions within the *server*, so no functional distribution will arise.

In general, distributed technical transactions can be avoided in all three design approaches. But modelling a control flow instead of a data flow bears a higher risk of introducing distributed transactions within the application landscape, due to the usage of synchronous communication.

None of the design approaches specifically supports the construction of an efficient message field structure nor prohibits the introduction of content based constraints.

2) Performance: Obviously, interfaces must satisfy the required performance, i.e., they must be able to deal with the given quantity description. Otherwise, the business process will not work correctly since required business functions may not be executed in time. Performance is supported by designing minimal interfaces with respect to the number of messages and avoiding synchronous communication [3], [15].

The more abstract the interface is, the less messages are needed due to the restriction of transmitting core concepts only. With a CRUD based design, the most abstract design is chosen while a use case based design includes relatively less functional abstraction.

Asynchronous communication is usually directly supported in the business process based design, while the other two approaches support a rather synchronous communication style. This holds especially for the CRUD based design, where the `read()` operation always enforces synchronous communication.

3) Understandability: Well designed interfaces must have a strong and documented relation to the underlying business context [1] thus ensuring a good usability of the interface. This will enhance the cost efficiency of the interface over time since a much better acceptance of the interface within the development teams will arise because the interface will be easier to learn, remember and use correctly [1].

Understandability is given by a strong functional binding between the business model and the implementation [16], the usage of business objects and business events as message content [3], [15] and a meaningful message naming schema.

Naturally, a business process based design leads to a direct mapping between interface and business process description thus enriching the interface with a comprehensive business context. On the contrary, a CRUD based design bears no business context at all due to its high level of abstraction.

Although all three approaches directly support the exchange of business objects, differences occur considering the publication of business events. A CRUD based design supports none of them per se, i.e., this approach forces a mapping of business events onto business objects. This will lead to serious problems in understanding the dynamical behavior of the application landscape. Using a business process based design instead, the published business events can be directly derived from the underlying business process. In contrast, a use case based design does not primarily focus on business events but on individual user operations thus obscuring the business context.

While the business process and the use case based designs both support message naming schemas providing a rich functional context, a CRUD based design uses only the given names for create, read, update and delete messages.

4) Recommendation: Considering the above mentioned desgin goals and the important advantage of supporting a direct binding between business model and interface design, the business process based design is the recommendated design style for interfaces, leading to the best design compromise.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

IV. Interface Operations

Complex application landscapes require the rollout of interface changes without shutting down all applications. To achieve this goal interfaces must be versioned and deployed during runtime, using the migration patterns described below.

A. Interface versioning

Every interface specification evolves over time due to syntactic, semantic or dynamic changes on the interface. These changes lead to different versions of the interface specification which are not compatible to each other. Therefore, the implementing applications must implement the correct version of the interface specification. In a complex application landscape, this is a common situation [6].

In order to guarantee a unique identification of a specific interface occurrence over time, each individual interface occurrence must have a version number [6]. Any change on an interface leads to a new interface version [6]. This includes syntactical changes in any message, changes within the message sequence flow, i.e., all changes of the dynamic behavior, and changes of the semantic behavior. Even the obviously simple cases of adding either a field to an existing message or introducing a new message to an interface represents a semantical change of the interface. This requires compatibility of the receiving application with the new interface specification version. Otherwise, severe problems may arise, if, e.g., a client executes syntactical message checks based on a specific interface version.

B. Big bang migration pattern

The simplest approach of an interface migration is *big bang*, where all applications are shutdown, redeployed and restarted at the same time, resulting in

$$1 + c \tag{1}$$

migration steps, where c denotes the number of participating clients. In case of a fallback, the server and all clients must be redeployed again.

C. Client first migration pattern

Within this pattern, the migration path is dominated by the clients. Each client will be successively migrated onto a new version that can handle both interface versions in parallel, as shown in Figure 5. In steps 1 and 2, the clients are changed to support additionally the new interface specification version. In step 3, the server is merged to the new interface implementation. Steps 4 and 5 are necessary to remove the support of the previous interface specification version from the clients.

After finishing all client migrations, the server will be upgraded to support the new interface version. Afterwards, all clients will be updated a second time in order to remove the support of the old interface version. During steps one to four of this migration path, the server will receive messages with a wrong interface version that must be ignored by the *server*.

The *client first migration* pattern will result in

$$1 + 2 * c \tag{2}$$

deployments, where c denotes the number of clients connected to the server. An advantage of this migration path is that

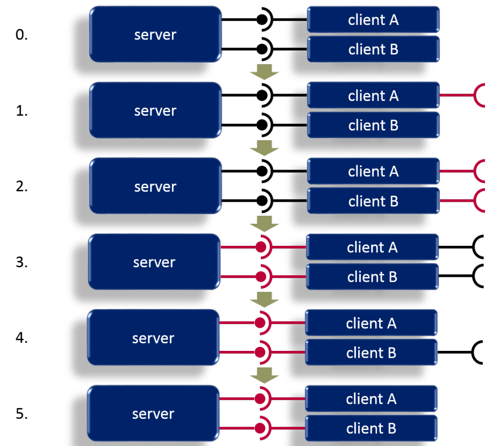

Figure 5. Steps of the client first migration pattern. The new interface version is denoted red.

clients can be upgraded independently from each other, i.e., no temporal coupling of the individual client migrations exist.

The price for this migration behavior is the necessary number of deployments : each client must be deployed two times, while the server is deployed only once. Furthermore, in case of a failure, the operational safe position of step 2 must be reached again. This is done by falling back with the server supporting the old interface version only and all clients whose support of the old interface version has been removed so far, requiring

$$1 + c_+ \tag{3}$$

steps, where c_+ denotes the number of clients migrated after the server migration.

D. Server first migration pattern

In contrast to the *client first migration* approach, the migration path can be reversed resulting in a server migration first followed by client migrations, see Figure 6. At step 1, the server provides support for two interface specification versions. In steps 2 and 3, both clients are merged successively. Finally, support of the previous interface specification version is removed from the server implementation, resulting in

$$2 + c \tag{4}$$

deployments. Again c denotes the number of participating clients. The advantage of this pattern is, that the number of deployments is

$$(1 + 2 * c) - (2 + c) = c - 1 \tag{5}$$

less than with the *client first migration* pattern. Note that during steps one to three of the migration path both *clients* A and B will receive invalid messages, which must be ignored, due to the concurrent interface version support of the server.

If a failure on the interface occurs within the migration path, all clients upgraded so far must fall back onto the previous interface version using c_+ rollout steps, where, again, c_+ denotes the number of clients migrated after the server migration. Thus, the operational safe position of step 1 is reached again.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-497-8

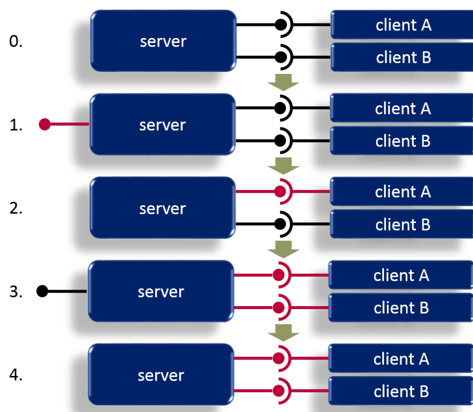

Figure 6. Steps of the server first migration pattern.

E. Comparison

The main differences between the migration patterns are the number of rollout and fallback steps and the required support of multiple interface versions within the applications. Beside the advantages of a lacking necessity to support multiple interface versions and a minimal number of rollout steps, the *big bang* pattern bears a high risk during fallback situations where multiple applications must fallback in parallel. Therefore, this pattern is only recommended if the number of clients is very small and a simultaneous fallback is organizational manageable.

Considering the other strategies, both migration patterns reduce the risk involved with a possible fallback compared to *big bang* at the cost of some additional rollout steps. Since the *server first migration* pattern requieres less rollout and fallback steps than the *client first migration* pattern, it is the recommended rollout strategy.

V. Summary

Due to the growing distribution of business functionality, interfaces have became very important for the behavior of an application landscape. Badly designed interfaces have a critical impact on the functional and operational behavior. To overcome these problems, this paper presented a structured and holistic approach of handling interfaces during design, build and runtime as follows.

Interfaces serve as contracts between applications. Thus it is inevitable to define the artifacts *message description*, *dynamic description*, *semantic description*, *infrastructure description* and *quantity description* to properly describe an interface with respect to the different aspects. In order to construct an interface, different design approaches have been presented and compared to each other. It turns out that the *business process based design* approach is most likely leading to the best result with respect to robustness, performance and understandability. Finally, different migration patterns have been presented introducing a new interface version into production environment. Due to the minimal number of required fallback steps in case of a severe error and one additional rollout step compared to the *big bang* pattern the *server first migration* pattern is recommended, at least for larger application landscapes.

Acknowledgment

The authors would like to thank their colleague Christian Wolf for valuable comments.

References

[1] M. Henning, “API Design Matters,” ACM Queue Magazine, vol. 5, 2007.

[2] J. Bloch, “How to Design a Good API and Why it Matters,” 2006, URL: http://landawn.com/How to Design a Good API and Why it Matters.pdf [accessed: 2016-06-08].

[3] R. J. Wieringa, Design Methods for Reactive Systems. Morgan Kaufmann Publishers, 2003, ISBN: 1-55860-755-2.

[4] M. Nygard, Release It!: Design and Deploy Production-Ready Software. O’Reilly, Apr. 2007, ISBN: 978-0978739218.

[5] U. Friedrichsen, “Patterns of Resilience,” 2016, URL: http://de.slideshare.net/ufried/patterns-of-resilience [accessed: 2016-06-06].

[6] B. Bonati, F. Furrer, and S. Murer, Managed Evolution. Springer Verlag, 2011, ISBN: 978-3-642-01632-5.

[7] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification,” The Internet Society, Specification, 1998.

[8] “Blink Protocol,” 2012, URL: http://blinkprotocol.org/ [accessed: 2016-06-06].

[9] “Financial Information eXchange,” 2016, URL: https://en.wikipedia.org/wiki/Financial_Information_eXchange [accessed: 2016-06-06].

[10] “FAST protocol,” 2016, URL: https://en.wikipedia.org/wiki/FAST_protocol [accessed: 2016-06-06].

[11] “Service-oriented architecture,” 2016, URL: https://en.wikipedia.org/wiki/Service-oriented_architecture [accessed: 2016-06-08].

[12] R. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” dissertation, University of California, Irvine, 2000.

[13] H. Kerner, Rechnernetze nach ISO-OSI, CCITT. H. Kerner, 1989, ISBN: 3-900934-10-X.

[14] “Client-server model,” 2016, URL: https://en.wikipedia.org/wiki/Client-server_model [accessed: 2016-03-03].

[15] G. Hohpe and B. Woolf, Enterprise Integration Patterns. Addison-Wesley, 2012, ISBN: 978-0-133-06510-7.

[16] E. Evans, Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004, ISBN: 0-321-12521-5.

[17] R. Westphal, “Radikale Objektorientierung - Teil 1: Messaging als Programmiermodell,” OBJEKTspektrum, vol. 1/2015, 2015, pp. 63–69.

[18] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2001, ISBN: 978-0-201-70225-5.

[19] B. Oestereich, Objektorientierte Softwareentwicklung: Analyse und Design mit der UML 2.0. Oldenbourg, 2004, ISBN: 978-3486272666.