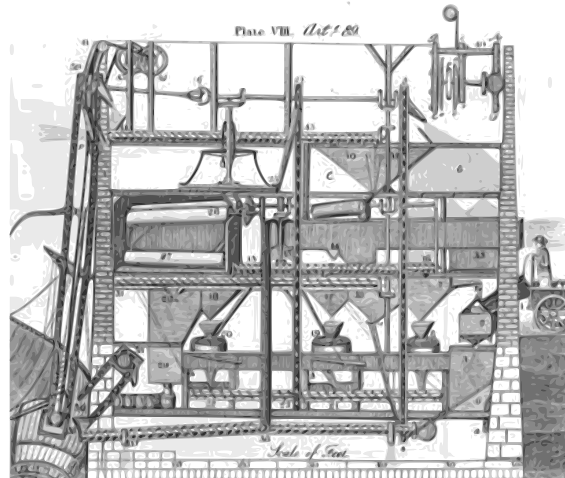
Invocation Assembly Lines

Patterns of Invocation and Message Processing in Object Remoting Middleware

Stefan Sobernig[1] and Uwe Zdun[2]
[1]*Institute for Information Systems and New Media*
Vienna University of Economics and Business (WU Vienna), Austria
stefan.sobernig@wu.ac.at
[2]*Distributed Systems Group, Information Systems Institute*
Vienna University of Technology, Austria
zdun@infosys.tuwien.ac.at

Object remoting middleware greatly facilitates creating distributed, object-oriented systems. However, developers face many situations in which a middleware's invocation and message processing architecture fails to fully support all their requirements. This problem is caused, for instance, by limitations in realising certain invocation styles (e.g., one-way and two-way conversations) on top of a shared processing infrastructure, in adding extensions to invocation handling (i.e., add-on services such as security and inspection), and in bypassing selected steps in the invocation handling to balance resource consumption and invocation performance. Often, these limitations are caused by design and implementation decisions taken early when crafting the middleware framework. To better explain the needed decision making, and help developers to apply adaptations or guide the selection of alternatives, we present a pattern language that captures the essentials of invocation and message processing in object remoting middleware. We also outline instantiations of the patterns and their relationships in existing middleware frameworks.



Evans' improved mill [22, Plate VIII] advanced the automation of pre-industrial further processing and refinement of wheat into flour. It is commonly regarded as an early predecessor of Fordist and more recent assembly line or production flow systems [44].

Copyright retain by author(s). Permission granted to Hillside Europe for inclusion in the CEUR archive of conference proceedings and for Hillside Europe website

1 Introduction

Developers who design and implement object remoting middleware are confronted with the task of supporting many different kinds of remote invocations. On the one hand, this variety is caused by the architectural setting in which the middleware is deployed and used. On the other hand, realising different kinds of remote invocations affects many spots of your middleware framework's design and implementation. These two issues are the root of two challenging design problems for the messaging processing infrastructure of a middleware.

The first design problem is the issue of *role distribution* (see e.g. [67, 63, 52, 3]): Applications built on top of your middleware framework must often play different roles, for example, they are expected to act both as a client issuing and as a server performing remote invocations. As these applications integrate the facilities offered by your middleware framework, your framework must facilitate the adoption by client- as well as server-side applications.

The second design problem is that there are a number of *crosscutting* concerns to be considered in your middleware framework design (see e.g. [34, 54, 70]): First, many application scenarios for your middleware require different remote invocation styles, component interaction styles, inter-component dependencies, levels of communication coupling, and so on. Second, many add-on tasks might be required, such as security or inspection add-ons. Finally, certain application scenarios are better addressed if your framework provides the flexibility to selectively omit certain steps in handling remote invocations. For example, optimisations may be applicable if signatures in interfaces of remote objects change frequently. This usually requires *bypassing* certain steps in the default control and data flow of your middleware framework. Each of these aspects, i.e., add-on services, component interaction and invocation styles, and bypassing, requires the interplay of different parts of the framework. Therefore, when developing your framework, you must anticipate a certain flexibility across essential building blocks of your framework.

This paper documents established design practises for realising versatile message processing infrastructures for remote invocations in object remoting middleware. The identified design practises set out to tackle the tensions caused by the crosscutting concerns of invocation handling and the issue of role distribution. We mined the design practises from existing object remoting middleware frameworks such as OpenORB [58], Mono .NET Remoting (Mono/R; see [40]), Mono Olive (Mono/O, [41]), Apache Axis2/Java (Axis2; see [5, 46, 21]), and Apache CXF (CXF; see [6]). We present our findings in terms of a pattern language and comment on the identified uses of the documented patterns.

This paper and the pattern language described herein are meant for those developers who must deepen their understanding of how object middleware frameworks can be designed in a manner to support varieties of remote invocations. The paper introduces the necessary terminology to foster your conceptual understanding of the inner workings of the middleware's invocation and message processing architecture. The background of this work are established remoting patterns – in the architectural context of the BROKER pattern [14, 32, 63, 55, 12]. The resulting pattern language provides links to existing remoting pattern collections [63, 12] and helps navigate in this body of established and documented design practises. An additional audience is the group of middleware users, who find structured guidance to evaluate existing object remoting middleware regarding its fit with requirements on supporting diverse invocation styles and on the middleware's extensibility. Finally, our pattern language targets developers who must further develop or adapt an existing middleware framework for complying with certain remote invocation types.

The paper is structured as follows. In the next section, we provide an overview of elementary terminology on object remoting and give some background on the BROKER pattern (see Section 2). Then, we introduce the reader to the pattern language as a whole and provide some hints on navigating between its patterns (see Section 3). Before giving the pattern descriptions themselves, we discuss the challenges of adaptable invocation and message processing (see Section 4). We provide a motivating

example in Section 4.2. In Section 5, the individual patterns of our pattern language are presented in detail. After having resolved the motivating example in Section 6, we describe known uses of the individual patterns in existing middleware frameworks (see Section 7). Finally, we conclude by discussing our pattern language and how it integrates with existing remoting patterns (see Section 8).

2 Some Background: Broker-Based Object Middleware

Assume you are using a middleware framework which follows the BROKER pattern [14, 32, 63, 55, 12], as it is the case for most modern object-oriented RPC middlewares, such as CORBA, .NET Remoting [40, 50], Windows Communication Foundation [41], and Web Services [9, 39, 16, 17] frameworks such as Axis2 [5] or CXF [6]. Figure 1 shows a set of basic remoting patterns from [63] and their interactions, forming a bare BROKER-based middleware framework. For a thumbnail overview of relevant object remoting patterns (and, for later reference) see Appendix A.

The exemplary BROKER is shown in the configuration for performing a remote invocation between a client component and a remote object: The client component performs an invocation on the remote object. Crossing the network boundary is handled by the middleware. That is, the REQUESTOR receives the invocation and constructs a request from the essential invocation data, i.e., the reference to remote object, the operation name, and the parameters. This results in a canonical object representation of the request. Then, the REQUESTOR uses a MARSHALLER to stream the objectified invocation data into a MESSAGE [29]. The MESSAGE is handed over to the CLIENT REQUEST HANDLER, along with the reference to the targeted remote object, for resolving the corresponding network endpoint, establishing a connection, and delivering this request MESSAGE. The MESSAGE arrives at the SERVER REQUEST HANDLER at the server side. Subsequently, the MESSAGE is forwarded to the INVOKER that initiates the disassembling of the MESSAGE. Disassembly involves demarshaling of the MESSAGE by a MARSHALLER. Demarshaling means to extract object references, contextual invocation data, and the core invocation data based on a canonical in-memory representation. Finally, the INVOKER resolves the remote object and dispatches the actual invocation. The invocation result is processed and returned in reverse order.

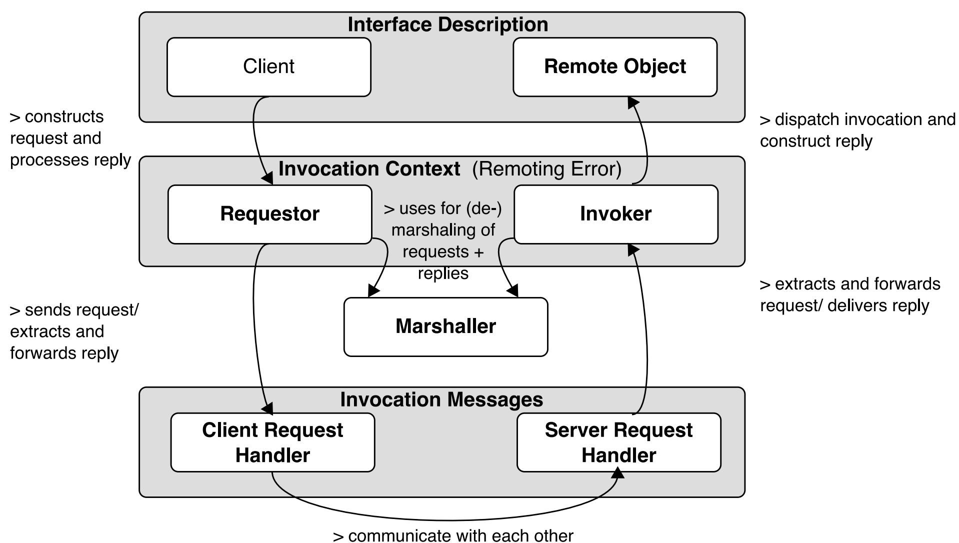


Figure 1: Relevant remoting patterns and their interactions

The structure of these remoting patterns and their interactions involve the processing and exchanging of invocation data items. This includes core invocation and result data (e.g., object references, operation names, parameters, object-type meta-data, MESSAGES [29, 12], and REMOTING ERRORS [63]), contextual invocation data (e.g., kinds of INVOCATION CONTEXTS [63]), and auxiliary invocation data

items which is not directly involved in a remote invocation (e.g., an INTERFACE DESCRIPTION [63]). In Figure 1, the invocation data items are represented by horizontal connectors that relate the instances of the remoting patterns operating on invocation data items at the same LAYER. For instance, the REQUESTOR constructs requests targeted at the INVOKER, and it processes reply objects generated by the INVOKER. The invocation data items involved are described in a number of remoting patterns. An overview of these patterns is provided by pattern thumbnails in Appendix A. More in-depth pattern descriptions and known uses are given in Appendix B.

3 Pattern Language Overview

This paper presents patterns focused on adaptable message processing infrastructures as they can be found in current object remoting frameworks, such as OpenORB [58], Mono .NET Remoting (Mono/R; see [40]), Mono Olive (Mono/O, [41]), Apache Axis2/Java (Axis2; see [5, 46, 21]), and Apache CXF (CXF; see [6]). The invocation and message processing infrastructures represent an integral extension mechanism of these middleware frameworks for realising support for add-on services, multiple invocation styles, and the selective bypassing of message processing steps.

The pattern language integrates the six patterns documented in this paper (see also Figure 2) with remoting patterns organised in two existing pattern languages: On the one hand, it extends the *Remoting Patterns* language documented in [63]. On the other hand, the patterns presented here link to the *Pattern Language for Distributed Computing* [12]. Whilst there is a certain overlap between these two pattern languages, in particular regarding the foundational BROKER pattern, each of these pattern languages contributes distinct patterns for better understanding of invocation and message processing. The Remoting Patterns contain extension and extended infrastructure patterns (e.g., INVOCATION INTERCEPTORS and CONFIGURATION GROUPS) which emphasise an adaptation view of a middleware framework. As a complement, the Pattern Language for Distributed Computing provides details on invocation data items processed, that is, kinds of MESSAGES. In this sense, this language stresses a data flow view of middleware frameworks. We aim at combining the adaptation and data flow views into a coherent description. Relevant patterns taken from these two pattern languages (and beyond) are presented as pattern thumbnails in Appendix A.

As a developer of a middleware framework, you usually foresee at least one of the following kinds of adaptability. In complex requirements settings, your framework must support all of them. The three kinds of adaptability correspond to the first three patterns in our pattern language:

- PARTIAL PROCESSING PATHS (see Section 5, pp. 15) are required to support different variants of one-way invocations. They allow you to lay out a complex processing scheme for invocation data and, in certain invocation scenarios, only to enact selected ranges of this scheme. This is needed for FIRE AND FORGET invocations, such as WSDL/1.1 one-way operations [16] or WSDL/2.0 in-only operations [17].
- RECONFIGURABLE PROCESSING PATHS (see Section 5, pp. 18) allow you to introduce additional processing operations, on demand and at arbitrary times in invocation processing. Thus, you can define a minimum processing scheme expected to be common to all or most invocation scenarios and permit extension developers to add processing operations as needed. Important examples of extended processing requirements are security-related, add-on invocation services, such as those described by the Web Services Security Core Specification (WSS/Core 1.0/1.1; [42]).
- PROCESSING SHORTCUTS (see Section 5, pp. 20) put you into the position to describe several possible walks through a processing scheme. While you lay out a basic scheme common to all or most invocations processed, you can still allow for deviating processing flows, for instance, by skipping a number of processing steps if required. This helps you to realise more complex forms

of exception handling, client- and server-side caching, redirecting invocations upon collocated remote objects, and so on.

Facing these requirements on adapting the message processing infrastructure, i.e., supporting previously unanticipated invocation styles, attaching add-on processing behaviour on demand, or bypassing of scheduled processing operations, you proceed by applying the INVOCATION ASSEMBLY LINE pattern. The INVOCATION ASSEMBLY LINE pattern describes your processing infrastructure in terms of *processing stations* and *processing tasks*. Processing stations denote elementary steps in the lifecycles of your REQUESTOR and INVOKER, or, similarly, the invocation data items processed. Invocation data items are characterised by certain processing states, e.g., `message constructed`, `message marshaled`, `message delivered`, and so on. Processing tasks describe processing operations performed on invocation data items in certain lifecycle states, e.g., a `marshaling` operation once entering the state `message constructed`.

Once you decided to adopt the INVOCATION ASSEMBLY LINE pattern, you face two further design problems:

- Is the number of processing stations fixed or adjustable?
- Do processing stations accept multiple task assignments or can each processing station only perform a single processing task?

The former design problem relates to designing and managing the *processing station layout* and the latter relates to the *station-task assignment*. In resolving these two issues differently, you have two choices for realising the INVOCATION ASSEMBLY LINE pattern:

- SINGLE-TASK PROCESSING STATIONS (see Section 5, pp. 22) describes the processing infrastructure by a number of processing stations and each of them is performing only a single processing task. In its extreme, you abandon the distinction of processing task and processing stations. The ultimate advantage of this variant is that there is no need for a dedicated, central management facility that allows you to map tasks to processing stations. You can adopt a decentralised organisation of invocation processing tasks.
- MULTI-TASK PROCESSING STATIONS (see Section 5, pp. 24), in contrast, allows you to assign multiple processing tasks to single processing stations. This involves applying absolute or relative ordering strategies for tasks attached to a single station. This variant puts you into the position of describing a common layout of processing stations, which is commonly fixed at design or configuration time, with distinct invocation scenarios resulting in different task configurations. The variable assignment often requires a central management of station-task mappings which entails changes at several spots (e.g., a manager and the station entities) to adjust the processing infrastructure to a new invocation scenario.

The patterns outlined above form a web of relations (see Figure 2). The INVOCATION ASSEMBLY LINE pattern is the *main* pattern of this language. It focuses on conceptualising and specifying invocation and MESSAGE processing. However, both the problem and solution of that pattern appear in a considerable number of variations so that we decided to treat these variations as distinct patterns. To begin with, we identified three *problem variants*: PARTIAL PROCESSING PATHS, RECONFIGURABLE PROCESSING PATHS, and PROCESSING SHORTCUTS. These three patterns vary from the main INVOCATION ASSEMBLY LINE pattern *by problem*, i.e., the INVOCATION ASSEMBLY LINE pattern is equally used to resolve three related, but sufficiently distinct problems. You may also think of use relationships between the three patterns and the INVOCATION ASSEMBLY LINE pattern. Similarly, there are two *solution variants*: MULTI-TASK PROCESSING STATIONS and SINGLE-TASK PROCESSING STATIONS. A

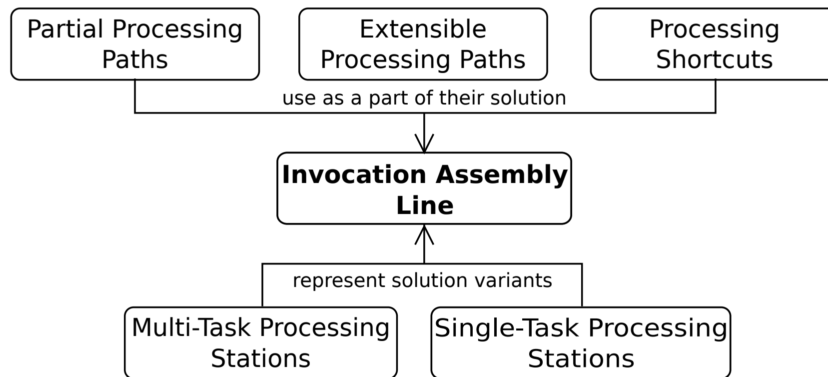


Figure 2: The patterns and their relationships

solution variant describes an alternative solution to the same problem. These variants share the problem stated in the main INVOCATION ASSEMBLY LINE pattern. Problem and solution variants form the two *fragments* [43] of our pattern language.

4 Challenges of Adaptable Invocation and Message Processing

The pattern language documented in this paper addresses situations in which a LAYERS-based view of a BROKER-based middleware framework (see also Figure 1; [14, 8]) turns out to be insufficient to effectively design and implement a middleware framework with particular extension capabilities. In such a view, the BROKER is segregated into layers of functional responsibilities under a rigid directionality. Components at higher-level layers, e.g., the client component, are constrained to use only functionality offered by components residing at their direct descendant layers, e.g., the REQUESTOR. As for crosscuts, applying modifications to components residing at all LAYERS or conditionally circumventing components in intermediate LAYERS is not considered. For instance, the client component is not expected to construct a request object on its own and hand it over to the MARSHALLER and CLIENT REQUEST HANDLER.

The LAYERS structure illustrated in Figure 1 also shows the client component and the remote object, as part of the server application, as conceptually separated. There is no notion of these two components exchanging their roles, so that the client turns into the server side (and vice versa). Also, since your framework is to be integrated by both client and server applications, it must integrate support for either side on a common ground. Otherwise, you risk introducing an unfavourable distinction between a *client-* and the *server-concrete framework* (see e.g. [52]), thus developing two sub-frameworks design- and implementation-wise. Such a distinction overlooks potentials for reducing design and implementation complexity by identifying shared characteristics in organising the processing of invocations and MESSAGES between, e.g., the INVOKER and the REQUESTOR components.

4.1 Realising Adaptability in Middleware Framework Design

Existing middleware frameworks provide variants of the INVOCATION INTERCEPTOR [63, 52, 53] pattern to extend this LAYERS structure by means of *indirection*: The INVOCATION INTERCEPTOR pattern [63] supports the definition of single processing operations that operate on invocation data items which are then transmitted using INVOCATION CONTEXTS [63] from the client to the server (and vice versa). INVOCATION INTERCEPTORS are registered with hooks placed within the processing infrastructure, for instance upon entering and upon exiting the MARSHALLER. Once reached, the hooks intercept the invocation data items processed (e.g., the request and reply objects) and have the processing operations defined by the INVOCATION INTERCEPTORS performed on them. The INVOCATION INTERCEPTORS

can be user-configured using CONFIGURATION GROUPS [63] that allow developers to define a number of related interceptors in a reusable group.

These patterns describe a common extension architecture, but they do not explain the internal mechanisms of the middleware to realise the composition of the processing tasks. In addition, not all processing tasks in middleware frameworks are INVOCATION INTERCEPTORS, and not all invocation data items are transmitted using INVOCATION CONTEXTS. For instance, some middleware frameworks define only the user-defined extensions as interceptors and use other mechanisms to define the basic processing tasks. It would be desirable both for the middleware designer and the middleware user to use one and the same internal mechanism to define and configure all processing tasks in a middleware.

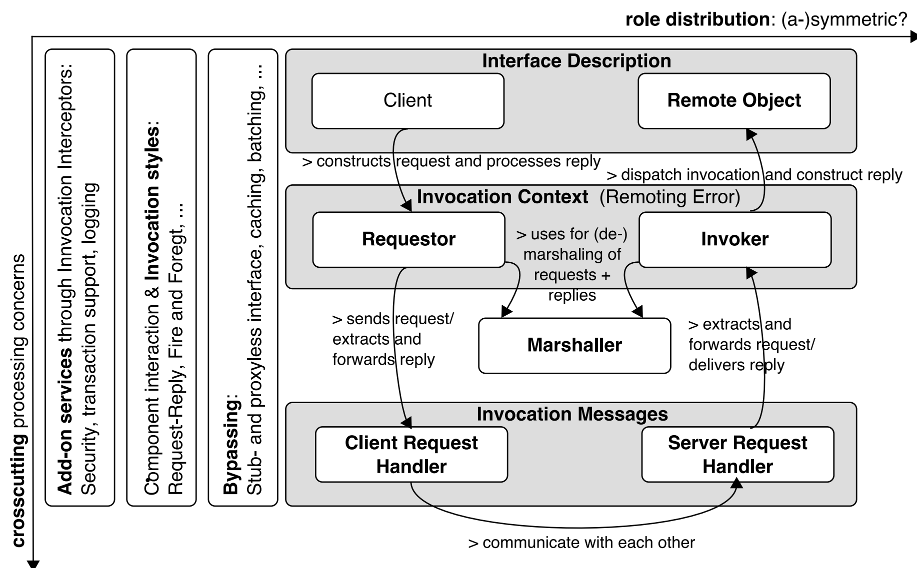


Figure 3: Requirement dimensions: role distribution and crosscutting processing concerns

There are two particular design problems related to finding such a canonical mechanism, originally discussed for the INVOCATION INTERCEPTOR pattern (see in particular [52]):

- *Hook selection*: At which LAYERS should hooks be placed? Also, at which spots should the invocation data items be indirected, e.g., on exiting, on entering, or somewhere within a given LAYER? The time of fixing this layout, e.g., the design, the configuration, or even runtime, is equally important. Providing for a wide and predetermined coverage through hooks risks causing substantial resource overhead and architectural complexity. Adopting a too limited number might prevent you or framework integrators from realising future extensions without modifying the basic INVOCATION INTERCEPTOR infrastructure.

- *Access protocol*: The access protocol regulates which elements of the invocation data items are exposed to and are made mutable by INVOCATION INTERCEPTORS. In addition, it regulates to which extent INVOCATION INTERCEPTORS influence the overall control flow. For instance, INVOCATION INTERCEPTORS might be allowed or disallowed to issue or process REMOTING ERRORS. The permissiveness of the access protocol balances the extensibility of the framework and the need for reliability when handling invocations on behalf of the middleware users. For instance, a too permissive access protocol might allow extension developers to manipulate core invocation data (e.g., the number of parameters), causing behaviour unexpected by the client application developer.

The pattern language presented in this paper helps you to understand and to address these two design problems. We focus on four classes of concerns which complicate the decision finding process due

to crosscutting the LAYERS structure of your middleware framework (see the vertical axis shown in Figure 3). They relate to components at several or all LAYERS: orthogonal add-on services, support for component interaction and their underlying invocation patterns, and kinds of LAYERS bypassing, and role distribution. The crosscutting structure of these concerns make it difficult to implement add-on services, different invocation styles, and kinds of bypassing in a fixed, foreseen way. Also, your framework design must be integratable by applications which take the client and server roles in turn. Following from this, the realisation of add-on services, invocation styles, and LAYERS bypassing must take into account framework support for both the client and server sides. We discuss this escalation in design complexity as the issue of role distribution (see the horizontal axis of Figure 3). A more detailed discussion of these four areas of challenge is provided in Appendix C.

4.2 A Motivating Example

Let us consider the situation shown in Figure 4 as an example. You need to secure remote invocations by means of encryption. There are many alternatives available to tackle such a requirement, most notably, transport-level encryption (for instance, TLS [20] or SSL [24]) and MESSAGE-level encryption (e.g., S/MIME; see [51]). However, both lack guarantees for secure *end-to-end* deliveries between a client and server application regarding transport intermediaries and MESSAGE authenticity. In addition, invocation data passes lower LAYERS of the BROKER unencrypted (see also Figure 1). Also, you are expected to add an invocation-level facility which permits client and server application developers to make use of *selective encryption and decryption* of MESSAGES sent and received. By selective, we mean that only parts of the MESSAGES are to be sealed, in particular the core invocation data. Data transmitted as the INVOCATION CONTEXT, which is also relevant for negotiating an en- and decryption scheme and routing MESSAGES, is to be left untouched. The requirements of end-to-end security and selectivity ruling out basic transport-level and MESSAGE-level options. Consequently, you must address this issue as an integral part of your invocation and MESSAGE processing infrastructure.

The UML activity diagram in Figure 4 presents a control and data flow view of the client side of processing invocations and MESSAGES (see also Figure 1). While the essential processing steps (e.g., `Construct request`, `Stream request`) are modelled as activities, activity partitions are used to identify the responsible remoting pattern for each of these steps (e.g., REQUESTOR, MARSHALLER). In addition, we represent data flow artefacts by means of input and output pins to activity nodes. For instance, `Invocation data` is the required input, and a `Request object` is the expected output of the `Construct request` activity owned by the REQUESTOR. Note that this activity diagram visualises the two-pass processing involved for the client side. That is, once the processing steps are performed on the request, and once on the reply (shown using the swimlane notation for activity partitions).

Given this control and data flow view of your given framework, you must decide how to realise the selective encryption facility, the `Security provider` (see Figure 4). This is essentially determined by the input required by such a facility, namely a streamed MESSAGE. The object representation of either request and reply are not suitable for applying encryption or decryption. This is mainly because their state is usually subject to further mutation and because they can't be properly constructed from the encrypted data enclosed by the MESSAGE. Therefore, you must provide means to operate on either the `Request` or `Reply message`, i.e., the output of the MARSHALLER for the outgoing request and the output of the CLIENT REQUEST HANDLER for the incoming reply. Which options are available to you?

1. Security-aware MARSHALLER: You might want to consider refining your MARSHALLER instantiation to perform the selective encryption. While this appears as a viable option at first sight, it would soon turn out impracticable because of code cluttering and constrained extensibility. Code cluttering would result from introducing conditional branching in the MARSHALLER to al-

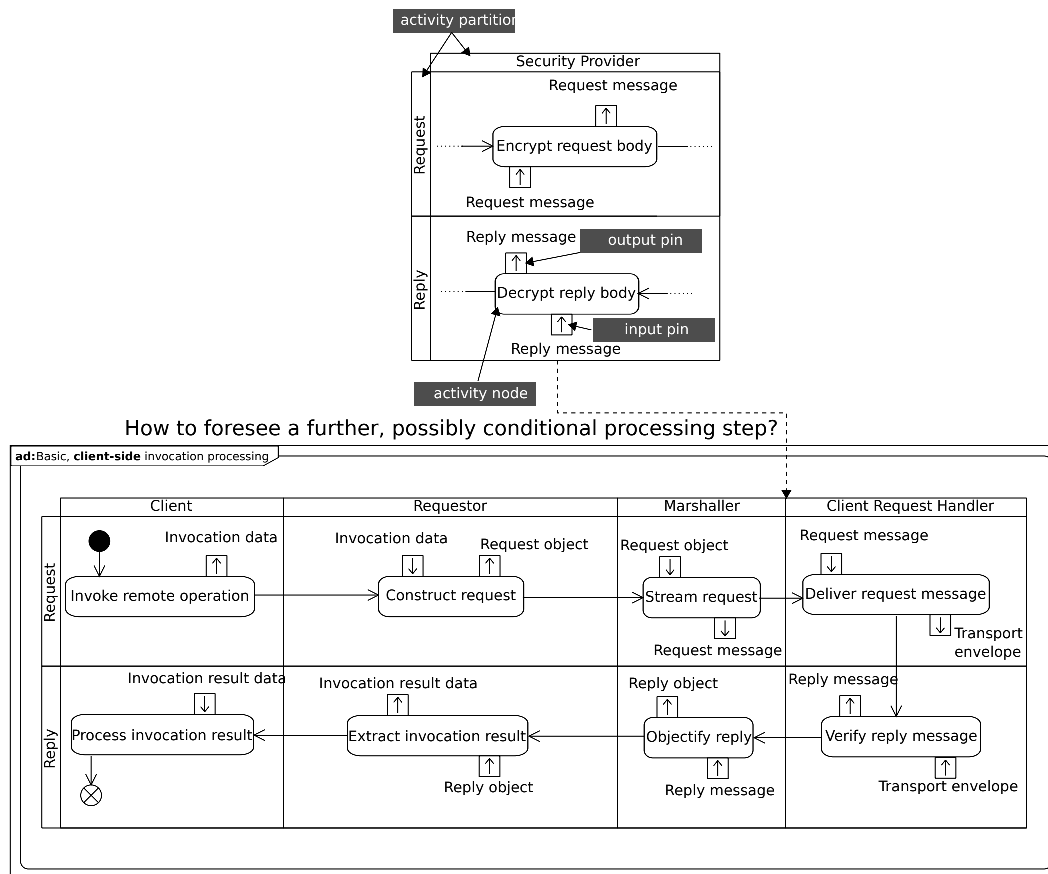


Figure 4: Example of an add-on service: Security provider for selective message encryption

low for selectively enabling or disabling encryption by clients. Extensibility and maintainability would be constrained if you supported a set of MARSHALLERS for different MESSAGE formats. In that case, future adjustments would have to be tracked by each of them.

2. Security-aware REQUEST HANDLERS: Alternatively, you could turn selective encryption into a responsibility of the CLIENT REQUEST HANDLER which has access to the required MESSAGE representations. Here, the critique put forth against the first, MARSHALLER-only refinement strategy applies as well. However, the situation turns out even worse: Given CLIENT REQUEST HANDLER variants for different transport protocols, you would also couple the transport to the message handling concern. This is due to the fact that the `Security Provider` requires intimate knowledge of the MESSAGE format (e.g., a SOAP/XML dialect) in order to select the parts meant to be encrypted or decrypted. Transport handling and transport protocol adoption (e.g., TCP [47], HTTP [23], or SMTP [48]) would be limited in their reusability. They would lose their independence from the MESSAGE format applied.

3. `Security provider` through INVOCATION INTERCEPTORS: Provided that your framework provides an INVOCATION INTERCEPTOR infrastructure (or, you plan to equip it with one), you can solve this extension problem by turning the `Security provider` into a set of INVOCATION INTERCEPTORS. You need at least two interceptors; one for providing the encryption and the other the decryption service. However, it is important that your INVOCATION INTERCEPTOR infrastructure is compatible with the specific requirements of a selective en- and decryption service (as shown in Figure 4). First, we require hooks placed at the end of the processing activities performed by the MARSHALLER or, alternatively, before the CLIENT REQUEST HANDLER. If either was missing, the `Security provider` would not be realisable by means of INVOCATION INTERCEPTORS because the MESSAGE representations needed could not be intercepted.

Second, the access protocol to the invocation data (i.e., the `Request` and `Reply messages`) must be permissive enough. For instance, if the access to and the mutation of the request and reply bodies was denied and only context data (e.g., for setting the encryption scheme and its details) was exposed to the INVOCATION INTERCEPTORS, realising the `Security provider` would not be feasible.

Given the three strategies outlined above, their inadequacies and possible constraints, what mechanisms remain open to you if you want to incorporate the `Security provider` into the processing control and data flow, as shown in Figure 4? When looking at a general-purpose extension infrastructure such as INVOCATION INTERCEPTORS, how can its design anticipate a wider range of extension requirements? How can it be made adjustable to fit those emerging in the continued lifecycle of your framework?

Not enough, this problem turns even more complicated. What if this addition must be configurable for different scopes, e.g., per-client or per-endpoint? How can you ensure that the addition of this encryption add-on preserves the modular organisation of REQUESTOR and CLIENT REQUEST HANDLER? Also, realising a `Security provider` both for client and server applications reinforces the problem. All this points to the issue whether the LAYERS structure still serves for the task of designing such an add-on service and the necessary framework facilities. This example reflects our motivation to mine existing middleware frameworks for the solutions adopted in response to more general classes of requirements on adaptable invocation and message processing.

5 Pattern Descriptions

Invocation Assembly Line

As an application developer, you use an existing BROKER-based middleware framework and you want to extend the message processing operations offered by this middleware framework. Or, as a framework developer, you design an extensible BROKER-based middleware framework. Extensibility is required to support application developers in adapting infrastructure for message processing to their application-specific requirements. This involves adding, removing, or replacing custom processing steps on various invocation data items. Certain kinds of invocation data items are found in almost all BROKER-based middleware frameworks, e.g., different types of messages, interface descriptions, etc. In addition, these invocation data items are subject to common processing operations, such as adopting a uniform object representation or their transformation into structured character or byte streams.

How do you extend a BROKER-based middleware framework with extra processing operations and application-specific refinements over invocation data items?

Functional extensions take either the form of application-specific ones or framework refinements as such, the latter being applicable to multiple, possibly related applications built on top of the middleware framework. To create such extensions, application and framework developers require access to the processing infrastructure for invocation data encapsulated by the INVOKER and the REQUESTOR. The processing infrastructure should be configurable and adaptable through a canonical programming model, at design time by the extension developer and at runtime by means of reflection, respectively. However, providing such a canonical programming model for configuring and adapting the processing infrastructure bears the risk of lacking the needed flexibility if the variety of invocation data items (e.g., MESSAGE kinds), the invocation patterns, the processing operations, and their interdependencies are not known when designing the middleware core. A threat to finding a balanced solution comes through the considerable variety of *processing operations* to be supported. This variety results both from different kinds of invocations to process and the processing range to cover.

To begin with, processing does not only refer to handling core invocation data. On the contrary, a BROKER must handle *auxiliary* kinds of invocations. These often require a very different processing scheme. Examples include the generation and delivery of INTERFACE DESCRIPTIONS (such as in Mono .NET Remoting) as well as certain auxiliary event MESSAGES. The latter commonly represent notifications orthogonal to the underlying remote invocation. In Web Services Reliable Messaging (WS/RM; [19]), for instance, message sequencing is implemented by particular notification messages which are exchanged completely transparent to the actual invocation messages. Also REMOTING ERRORS form a distinct group of invocation data to process.

When being processed, invocation data items pass different processing stages. When designing a programming model providing access to the processing infrastructure, it is difficult to decide which processing stage at which level of granularity should be incorporated into this programming model, in terms of extension points. Candidates are the REQUESTOR and INVOKER which describe their range of processing in terms of demarshaled and marshaled core invocation data and results, their delivery and reception, and so on. In addition, the CLIENT PROXY and groups of remote objects could be incorporated into the design of such an extensible processing infrastructure. The latter is exemplified by *contexts* in Mono/R which are controlled execution environments for remote objects.

The design task for a versatile processing infrastructure in your middleware framework is further complicated by the *processing roles* (simply *role* hereafter) to be supported. In different component interaction and invocation styles, the middleware framework needs to take different roles (e.g., the *client* or *server* role in a REQUEST-REPLY invocation). A processing role is described by a set of processing operations and a sequencing of these operations which are specific to this role. The number and

characteristics of processing roles escalate with each component interaction style, invocation pattern, and their variants specific to a remoting technology family supported. This motivates to identify similarities and points of variations between processing roles in terms of processing operations and their dependency relations.

You also risk introducing unwanted design complexity through *invocation style emulation*: This problem describes the situation when the design of your BROKER incarnation is centred around a predominant invocation pattern, e.g., REQUEST-REPLY. From the perspective of evolvability, adding support for e.g. FIRE AND FORGET means to build an additional MESSAGE processing scheme on top of a processing infrastructure aligned to the predominant pattern. This strategy of emulation has been reported to bear the risk of introducing complexity. This extra complexity is due to static and dynamic crosscutting. On the one hand, there is the risk of increased code interlacing (see e.g. [34, 54]). On the other hand, the additional need for conditional branching (see, e.g., [70]) complicates the control flow inherent to such a processing infrastructure.

Therefore:
Organise the invocation and MESSAGE processing of your middleware in terms of INVOCATION ASSEMBLY LINES. INVOCATION ASSEMBLY LINES are configurable and extensible chains of message processing tasks used both on the client (e.g., owned by the REQUESTOR) and the server side (e.g., owned by the SERVER REQUEST HANDLER). These processing chains are put in place for both request and reply MESSAGES. They are partitioned into processing stations which form the message processing flow. Each processing station contains one or more processing tasks in a specific order. The processing tasks are to be performed on certain invocation data items (e.g., MESSAGES or INVOCATION CONTEXTS) to be handled by the station. An INVOCATION ASSEMBLY LINE exposes a programming model to adjust the number of processing stations and offers different strategies to assign processing tasks to the available processing stations. INVOCATION ASSEMBLY LINES should be constructable at design and at activation time (e.g., through deployment descriptors), as well as changeable through the programming model at runtime.

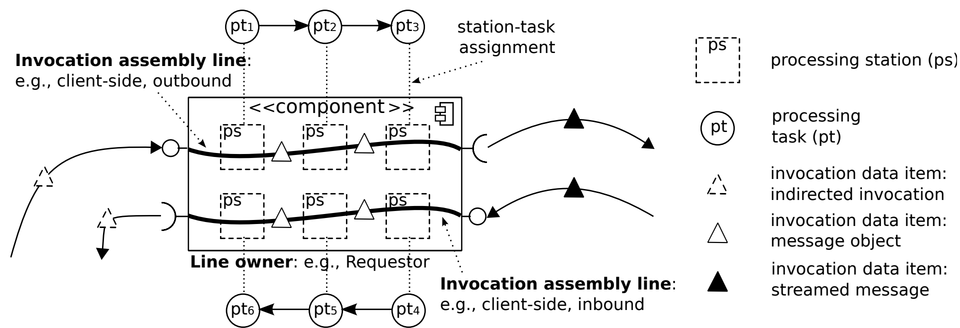


Figure 5: Conceptual sketch of INVOCATION ASSEMBLY LINES in a REQUEST-REPLY scenario

The INVOCATION ASSEMBLY LINE pattern describes the BROKER or a selection of its component patterns as a reconfigurable, flow-oriented processing infrastructure (see Figure 5). There are the following participants:

- **Processing station** (ps): An INVOCATION ASSEMBLY LINE consists of processing stations. They perform specific assembly, disassembly, and transformation operations on invocation data items. For each incoming or outgoing message (request or reply), invocation data items are transported along the INVOCATION ASSEMBLY LINE by passing them from station to station. Different stations can potentially process different kinds of invocation data items, The reconfiguration of an INVOCATION ASSEMBLY LINE means adding or removing processing stations

from a given configuration. In addition, a processing station can participate in several independent INVOCATION ASSEMBLY LINES. This permits us to create processing layouts beyond mere serial processing.

- **Processing task** (pt): Different kinds of invocation data items are subject to a set of related *processing tasks*. The tasks necessary to process these invocation data items vary considerably in different component interaction styles, dependency couplings, invocation styles, add-on infrastructure services, and remoting technology families. The kind of assembly, disassembly, or transformation operation described by a task is depending on the kind of invocation data item, such as core, contextual, or auxiliary invocation data. A particular processing task may be relevant for several MESSAGE kinds, possibly leading to reusing task descriptions and their implementations. Looking at the REQUEST-REPLY example in Figure 5, for instance, we find the processing tasks of marshaling and demarshaling the objectified MESSAGES.

  Processing tasks are interrelated by an explicit ordering. The example in Figure 5 shows, for instance, that a canonical MESSAGE object representation must be constructed by the INVOKER or REQUESTOR before being marshaled into the actual MESSAGE by the MARSHALLER. The ordering structure of processing tasks are represented as a graph. This graph is formed by the *pt*-labelled vertices representing processing tasks, while edges denote their directed dependency relations. These precedence structures often reveal a *symmetry* of tasks within a given processing roles or between two processing roles. Looking at Figure 5, the `marshaling` (pt_2) operation performed in the outbound direction mates with a `demarshaling` (pt_5) operation on incoming invocation messages.

- **Invocation data item**: An *invocation data item* represents a workpiece to be processed by the processing stations grouped into an INVOCATION ASSEMBLY LINE. Relevant invocation data items are the introspection data of the indirected invocation and the result. They are then extracted from or transformed into MESSAGES. Further examples are INVOCATION CONTEXTS and context-related MESSAGES. Furthermore, auxiliary invocation data kinds such as INTERFACE DESCRIPTIONS are important to consider. The REQUEST-REPLY invocation example in Figure 5 shows MESSAGES representing invocation requests and invocation replies in various processing states, i.e., runtime information about an invocation, an invocation object, and a streamed invocation message. In addition, REMOTING ERRORS must be processed, if they occur. They must be signalled between the remoting endpoints.

- **Line owner**: INVOCATION ASSEMBLY LINES can be applied to structure the processing activities described by certain component patterns of the BROKER. For example, INVOCATION ASSEMBLY LINES are used by the REQUESTOR and SERVER REQUEST HANDLER as a part of their solution. We refer to the using framework components as *line owners* in the collaboration described by the INVOCATION ASSEMBLY LINE pattern. Also, we find INVOCATION ASSEMBLY LINES applied to INVOKER implementations to keep the invocation dispatch mechanisms reconfigurable and extensible.

- **Binding scope and times**: Binding, in this context, refers to the scopes and the times for activating and deactivating a particular INVOCATION ASSEMBLY LINE configuration. An INVOCATION ASSEMBLY LINE configuration describes a number of processing stations and the processing tasks assigned to them according to their precedence requirements.

  Valid and relevant scopes for binding a processing configuration are single remote objects. If shared by many remote objects, the REQUESTOR and the INVOKER are appropriate scopes. A more general and reuse-driven approach is to bind a processing configuration to a CONFIGURATION GROUP. A CONFIGURATION GROUP [63] describes the configuration of the MESSAGE processing and the lifecycle management infrastructures shared by a set of remote objects. In addition, a CONFIGURATION GROUP can act as controlled execution environments for remote objects. Examples include *contexts* in Mono .NET Remoting (Mono/R; see [40]).

Besides the scope of enactment, various points in the lifecycle of a BROKER instance are candidates for defining the binding times to specify, activate, and deactivate a processing configuration. In the middleware frameworks reviewed, the deployment time of either remote objects or CONFIGURATION GROUPS has been chosen as the binding time. The issue of binding times also leads us to ask for an appropriate specification and deployment technique. Options include registration through the programming model at runtime (PASSIVE and ACTIVE REGISTRATION [37]) or by means of deployment descriptors [63] at activation time. The late negotiation and acquisition of such a configuration, i.e., at invocation time, can be a further requirement. For instance, invocation pattern variants (such as SOAP/1.2 or WSDL/2.0 Message Exchange Patterns, MEPs; [17, 25]) can be lazily negotiated between a client and a remote object. Either, the INTERFACE DESCRIPTION stipulating the invocation pattern is introspected just in time (e.g., in forms of dynamic invocation) or the invocation pattern is communicated through the INVOCATION CONTEXT. In either case, the INVOCATION ASSEMBLY LINE needs to be assembled and activated at runtime.

INVOCATION ASSEMBLY LINES organise the control and data flow for their line owners, such as the SERVER REQUEST HANDLER or REQUESTOR. There are two dimensions of control and data flow to consider: the *placement* and the *centralisation* of the control and data flow layout.

The *placement* of control and data flow information can either be extrinsic or intrinsic to the processing stations. An *extrinsic* placement refers to capturing the overall processing-related control flow information in dedicated runtime entities different to processing stations. In particular, relevant control and data flow information can be stored and managed by invocation data items. The INVOCATION CONTEXT is sometimes used for this purpose. An *intrinsic* placement challenges the above view by locating control and data flow information in the processing stations directly.

With regard to *centralisation*, the control and data flow layouts can be organised in a centralised or dispersed manner. A *centralised* INVOCATION ASSEMBLY LINE lays out the control and data flow in single, state-carrying entities different from the actual processing stations. For instance, the `AxisEngine` object-class as an essential component of this combined INVOKER and REQUESTOR variant in Axis2 stipulates the overall control and data flow centrally, in its `send` and `receive` operations. Conversely, a *dispersed* specification of the control and data flow allows for distributing responsibilities among several processing stations. In Mono/R, for instance, *sinks* realise this idea of dispersed control and data flow management for INVOCATION ASSEMBLY LINES. Each sink is only aware of its neighbour sinks, leaving the ultimate succession of sinks managed in a decentralised manner.

Combining these placement and centralisation strategies shows different effects on the targeted adaptability of processing behaviour, the locality of modifications and the perceived complexity of the processing infrastructure. This is particularly important when considering the need to provide introspective facilities upon the control and data flow to realise runtime and invocation time adaptability of INVOCATION ASSEMBLY LINES. Also, you may consider assigning one or multiple processing tasks to a given processing station. Different strategies of station-task assignments, placement, and centralisation as well as an assessment of their consequences are covered by the SINGLE- (pp. 22) and the MULTI-TASK PROCESSING STATIONS (pp. 24) patterns.

Partial Processing Paths

Assume that you have decided to build or adapt a certain BROKER incarnation. The kinds of remote invocation (e.g., REQUEST-REPLY and FIRE AND FORGET RPCs) and the remoting styles (e.g., functional subsets of ICE [27] and WS [9, 39, 16, 17]) which this BROKER realisation is meant to support are given. Assume further that your BROKER-based framework is intended to serve as a surrogate for both client- and server-side applications. In order to be sure that the processing of MESSAGES is suitable for serving the given set of invocation patterns for the range of remoting styles and roles (i.e., client and server) supported, you must decide how to lay out the MESSAGE processing infrastructure. Invocation patterns (e.g., REQUEST-REPLY, FIRE AND FORGET) do not reveal insights on the different processing needs of MESSAGES within an invocation pattern realisation. Processing operations and sequencing rules between single operations (e.g., marshaling, MESSAGE construction, etc.) could be reused for realising different invocation patterns. Others might be conflicting in the context of a particular invocation pattern.

Your invocation and MESSAGE processing infrastructure must allow to perform selected ranges of designed paths of processing operations alone. How can a processing infrastructure be designed out of a set of composable path sections which can be combined into a bound variety of processing paths?

Consider an example: A processing scheme originally designed to realise REQUEST-REPLY invocation variants might be required to put only the request-specific range into effect to enact kinds of FIRE AND FORGET invocations (see also Figure 6). If, in a FIRE AND FORGET scenario, reply-specific processing steps would be scheduled, the responsible entities would, at least, have to operate on the control flow in a manner to be effectively skipped. The risk of excessive conditional branching would be the result, to give a single concrete example. Besides, maintainability is potentially reduced because these decision points which implement the processing line in a requested invocation pattern are squattered over several implementation entities. More generally speaking, this compositional flexibility is demanded by different types of one-way invocation, such as forms of synchronisation decoupled and non-blocking invocations (e.g., FIRE AND FORGET) as well as timely decoupled one-way acts in MESSAGING and PUBLISH-SUBSCRIBE [12].

Also, you might want to consider support for batch processing of invocations [59]. To realise batching support, you must define a processing scheme which permits the repetitive execution of single processing operations early in the lifecycle of remote invocations. Either the CLIENT PROXY or the REQUESTOR must be capable of accumulating invocation dispatches. Accumulation involves the incremental assembly and later disassembly of batch MESSAGES. Only upon signalling the end of the batch invocation (e.g., through an explicit *flush* operation), the REQUESTOR proceeds in further-processing the accumulated MESSAGE, i.e., its marshaling and delivery. In an inverse manner, the INVOKER has to disassemble the batch MESSAGE, perform the individual invocation dispatches, compile a reply MESSAGE, and, finally, have it streamed and returned. Hence, batching causes single processing steps to be repeatedly performed in a row (e.g., MESSAGE construction) while others are only scheduled once (e.g., marshaling).

In addition, you are often faced with the requirement of handling and even recovering from REMOTING ERRORS [63] as well as exceptional conditions when processing MESSAGES. This very requirement can be raised from different angles:

1. To begin with, REMOTING ERRORS can be raised at different locations (e.g., the client, server application or the transport logic). In particular, they can be issued from within the MESSAGE processing infrastructure, for instance, upon sensing exceptions when turning a MESSAGE object into a structured byte stream representation (e.g., an IIOP [45] message). Depending on the concrete type of REMOTING ERROR, they must be propagated back to the remote endpoint, i.e., the client application or its underlying BROKER, where the encapsulated REMOTING ERROR is

injected into the control flow. This involves processing REMOTING ERRORS as MESSAGES to be delivered to the remote endpoint. The processing steps for these REMOTING ERRORS and their sequencing are essentially similar to those of core invocation data (i.e., the request and reply data).

2. Forms of reliable messaging (e.g., the error recovery model in ICE [27] or WS/RM [19]) presuppose that the MESSAGE processing infrastructure can automatically replay sequences of processing steps to realise quality constraints on MESSAGE delivery, e.g., at-most-once delivery guarantees. Replaying refers to performing varying ranges of processing steps depending on the point of exception or failure, possibly in a repeated manner.

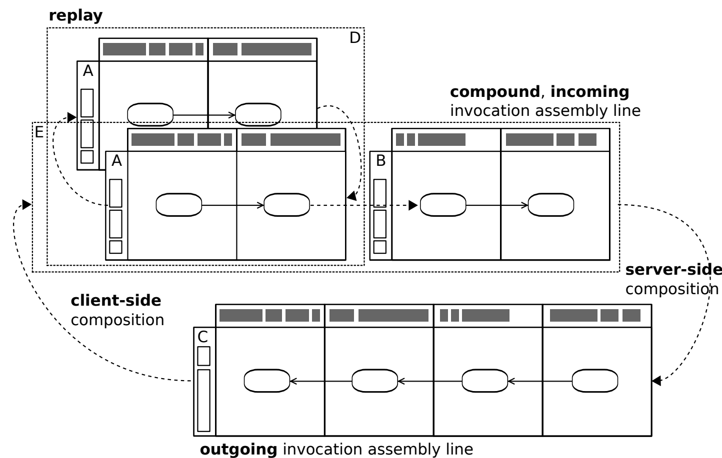


Figure 6: Composable processing of invocations

Therefore:
Organise your MESSAGE processing infrastructure in terms of PARTIAL PROCESSING PATHS. Each PARTIAL PROCESSING PATH is realised by a dedicated INVOCATION ASSEMBLY LINE representing a set of functionally linked processing operations which recurs in different targeted invocation pattern realisations. As INVOCATION ASSEMBLY LINES are joinable, combine their instantiations for expressing different processing roles (client or server side) and invocation pattern implementations.

PARTIAL PROCESSING PATHS promote the idea of structuring the processing infrastructure in smaller parts which can be reused by composing the actual processing paths as compounds from such building blocks. Each part groups processing tasks (e.g., marshaling, decryption, etc.) which are found recurring jointly in different invocation scenarios. By joint recurrence, we mean that they are applied in a sequence which is found stable for a number of invocation scenarios. Each part is represented by a dedicated INVOCATION ASSEMBLY LINE. An INVOCATION ASSEMBLY LINE can take the form of a compound (e.g., line E in Figure 6), assembled from other INVOCATION ASSEMBLY LINES (e.g., lines `A` and `B` in Figure 6). While the number of possible scenarios is certainly vast, you will find certain, characteristic decomposition strategies across current middleware examples:

- *Per processing direction*: Provide INVOCATION ASSEMBLY LINES which represent processing directions, i.e., outward- and inward-directed invocations and MESSAGES. In Figure 6, there are the outgoing line `C` and the incoming line `E`. Such a decomposition strategy makes it convenient to express the client and server roles in the light of a given invocation patterns. As for REQUEST-REPLY invocations, the composition set (`C,E`) represents the client-side, the set (`E,C`) the respective server side of processing. If FIRE AND FORGET is requested, the processing infrastructure is limited to the line `C` at the client and the line `E` at the server side.

- *Per kind of invocation data*: Certain processing tasks appear shared between different types of invocation data items, while others are type-specific. Sets of shared processing tasks are then composed into compound INVOCATION ASSEMBLY LINES to handle a specific type of invocation data item. Important examples are REMOTING ERRORS, INTERFACE DESCRIPTIONS, and certain auxiliary MESSAGE kinds underlying advanced invocation patterns (e.g, notifications in WS/RM [19]).
- *Per lifecycle strategy*: Certain sections of the processing path are performed repeatedly. This is particularly important for strategies of failure recovery and invocation batching. The repetitive character of a set of processing steps qualifies these to form INVOCATION ASSEMBLY LINES which then are grouped into compound lines. In Figure 6, the component line A provides an example in the context of incoming invocations requests. This maps to a server-side infrastructure which is capable of recovering from errors encountered in early processing steps (e.g., transport and demarshaling).

Use PROCESSING SHORTCUTS (p. 20) to provide the links between the resulting INVOCATION ASSEMBLY LINES. For example, INVOCATION ASSEMBLY LINES operating one core invocation data should include PROCESSING SHORTCUTS to those responsible for possible REMOTING ERRORS. Make sure that the decomposition into PARTIAL PROCESSING PATHS is not hindered by a lack of genericity of the components realising the actual processing tasks. Consider a MARSHALLER which is bound to the canonical object representation of core invocation data (i.e., request and reply objects) and which is only capable of processing these. As a result, you cannot create a PARTIAL PROCESSING PATH to be reused for various data kinds (e.g., INTERFACE DESCRIPTIONS).

Reconfigurable Processing Paths

When you plan to provide framework extensions to a middleware, it is important to verify that you can actually weave the extension behaviour into the framework and that the framework provides the necessary extension points to realise the add-on behaviour. Similarly, if your role is that of the framework developer, you will certainly find yourself in the situation that prospective extension developers express requirements on extension points to be exposed by the processing infrastructure. It is particularly hard to foresee the number and kind of extension points becoming necessary at the time of creating the processing infrastructure. Each step in a processing path can be turned into an extension point by refining it into a hook for INVOCATION INTERCEPTORS [63, 52]. However, not each processing step should be fixed as a hook. This patterns helps avoid predetermining a web of hooks at design time. Such a predetermination bears the risks of excessive resource consumption and an increased design complexity which reduces the communicability of the hooks and their interdependencies to extension developers.

Providing support for extensions and add-on services plays a central role for the adoption and the further-development of a middleware framework. Unless you can add or remove extension points to the processing infrastructure for invocations and MESSAGES in a principled manner, the middleware will not fit unanticipated deployment scenarios while preserving its maintainability.

A middleware framework, as a variant of an object-oriented application framework, is not meant to be a ready-made piece of software. Rather it is to be completed through integration by client and server applications. Integration also means to attach framework extensions, shared by a group of client or server applications. Designed processing paths must remain adaptable to cover a modified set of processing operations and changing dependency relations between them. This is particularly important regarding your framework's extension infrastructure which might be built around an INVOCATION INTERCEPTOR [63, 52] variant. From this perspective, processing operations are potential points of interception. In order to extend (or reduce) the INVOCATION INTERCEPTOR's reach, processing steps must be addable (or removable). Important examples are found in securing remote invocations because security-related, orthogonal extensions such as implementations of the Web Services Security Core Specification (WSS/Core; [42]) commonly operate on MESSAGES before and after core processing steps, such as demarshaling and marshaling.

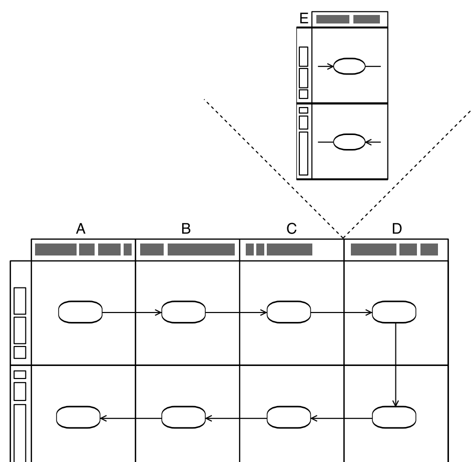


Figure 7: Extending processing paths

Therefore:
Preserve RECONFIGURABLE PROCESSING PATHS by realising each processing path as an INVOCATION ASSEMBLY LINE. Allow to insert into or remove processing stations from these INVOCATION ASSEMBLY LINES. Alternatively, if the layout of processing stations is fixed, make sure that

processing stations can either be effectively discharged from all processing tasks (i.e., a de facto removal) or that they can be attached more than one processing task (i.e., a de facto insertion).

At design time of your framework, make sure that you apply the PARTIAL PROCESSING PATH pattern to obtain an initial set of relatively robust INVOCATION ASSEMBLY LINES. They can then be exposed for refinement into RECONFIGURABLE PROCESSING PATHS. The sketch provided in Figure 7 shows a compound of two INVOCATION ASSEMBLY LINES to realise a client-side processing configuration for a REQUEST-REPLY invocation variant. In its initial configuration, there are four processing steps (i.e., `A,B,C,D`) and, thus, candidate extension points. By adding a processing station to each line, you obtain the further processing step `E`. Note that this exemplary insertion preserves the processing symmetry [63, 52], by considering two symmetric processing stations. This is, however, not strictly necessary. It would be equally possible to amend only one of the two INVOCATION ASSEMBLY LINES in Figure 7. For instance, if only logging of outgoing invocation data items was required, this would suffice as an extension point.

Arrange the protocol to reconfigure the processing paths in a way that adaptations to the processing path can be performed by the extension developers themselves. This avoids conflicts between reconfiguration requirements of different extensions. This requires respective hooks being activated and deactivated at configuration and runtime. Details for these issues of binding scope and binding time are treated by the INVOCATION ASSEMBLY LINE pattern (pp. 11).

The need for RECONFIGURABLE PROCESSING PATHS is exemplified by the scheduled design element of *dynamic phases* in Axis2 [5]. So far, Axis2 extension developers cannot revise and introduce their own set of phases, i.e., processing steps, through their CONFIGURATION GROUPS contributed. This, however, turned out critical because the global phase configuration must be adjusted or various variations thereof need to be shipped in order to support individual extensions. This breaks the fundamental idea of orthogonal extensibility and introduces an unwanted coupling between the core framework and its extensions. The planned design revision will allow for per-extension phases to overcome this limitation [30].

Processing Shortcuts

Within processing paths – as yielded by applying the PARTIAL PROCESSING PATH (pp. 15) – there are situations which either demand an interruption of the processing sequence or rather skipping subsequent processing steps. To complicate things, these situations are sometimes only identifiable at runtime. Skipping does not necessarily require these processing steps to be circumvented. However, the input and output requirements for the follow-up processing steps might not be met and, thus, special care would be required, e.g., by providing mock entities. While the interruption case refers to the occurrence of REMOTING ERRORS, the skipping requirement relates to forms of bypassing in the LAYERS structure of your processing infrastructure. This pattern describes the details of splitting predetermined processing chains and of laying out alternative processing walks.

Processing paths describe sequences of processing operations applied to invocation data items. These processing chains are the most natural way to think of your processing infrastructure. However, certain optimisations towards resource consumption, invocation performance, and decoupling between remote ends are impossible to achieve with such processing chains in place.

A designed processing layout breaks down into a single dominant processing path for invocation data. However, there are scenarios which require multiple possible processing paths by mutating the sequencing of a given set of processing steps. Relevant examples are:

- MESSAGE caching and differential marshaling (see e.g. [4, 1]): These optimisations aim at reducing the time spent in the MARSHALLER, either by caching MESSAGES (or their object representations) entirely or partially. As for partial caching, only the meta-data specific to the MESSAGE structure is put into a cache storage. This allows for applying caching though the invocation data constantly change.
- Handling exceptional values, in particular REMOTING ERRORS [63]: Upon the signalling a failure condition, e.g., through the exception propagation mechanism of the hosting runtime environment, the processing is expected to be forward-skipped (e.g., to cleanup resources already allocated such as connections etc.) or control is handed over to a processing path specific to REMOTING ERRORS (e.g., to marshal and deliver them to the remote end).
- Invoking upon *collocated* remote objects (see e.g. [61]): The remote object targeted lives at the same remote end as the issuing client, though not necessarily in the same process habitat. In these situations, the majority of core processing steps (e.g., objectifying requests, marshaling, transport) can be considered pure overhead and are to be avoided.

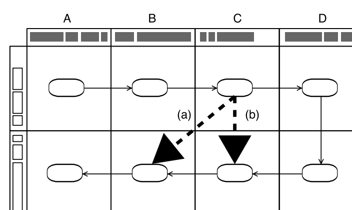


Figure 8: Permitting multiple, conditional processing walks

Therefore:
Provide PROCESSING SHORTCUTS for INVOCATION ASSEMBLY LINES which can redirect the control and data flow from one processing station to another one. This target processing station can be located in the same INVOCATION ASSEMBLY LINE, e.g., further down the line so that intermittent processing steps are skipped. Also, the target processing station can be part of another INVOCATION ASSEMBLY LINE. So, you can redirect the control and data flow between the two

lines and you can have the receiving line continue (or even complete) the invocation and message processing.

The requirement of bypassing is realised by redirecting the control and data flow between two INVOCATION ASSEMBLY LINES. Details on how PROCESSING SHORTCUTS can be laid out and how these line-crossing processing stations can be constructed are dependent on the realisation variant of the INVOCATION ASSEMBLY LINE pattern used (see also Figure 8): If the layout of processing stations is global and fixed during runtime (i.e., the MULTI-TASK PROCESSING STATION pattern applies; pp. 24), PROCESSING SHORTCUTS can bridge between different processing steps. Consider the example in Figure 8: The outgoing activity of processing step `C` could redirect to the incoming activity of processing `B`. This is because processing stations can be made aware of each other. Conversely, if the station layout is not predetermined (i.e., the SINGLE-TASK PROCESSING STATION pattern applies; pp. 22), a cross-line station can only be created within a single processing step. Hence, the outgoing activity can only point to the incoming activity in step `C` (see Figure 8).

Use this pattern jointly with PARTIAL PROCESSING PATHS (pp. 15) to provide the necessary connectors between INVOCATION ASSEMBLY LINES specific to a processing direction and to a type of invocation data. The PROCESSING SHORTCUT pattern implies that, when applied along with RECONFIGURABLE PROCESSING PATHS (pp. 18), inserted processing steps are to be symmetric.

Single-Task Processing Stations

You chose to adopt the INVOCATION ASSEMBLY LINE (pp. 11) pattern. It remains to select a strategy for laying out the processing stations and the assignment of processing task to these stations. This strategy is largely about where to place the control for the station layout and the task assignments; *and* whether the control and data flows are centrally managed. Your main objective is to ease the future addition of framework extensions (e.g., an encryption add-on) which come in form of CONFIGURATION GROUPS. Your middleware framework is required to facilitate the extensibility towards orthogonal add-on services (e.g., for securing the BROKER). Framework extensions risk entailing hidden interdependencies, when being deployed together, which can cause single extensions to fail unexpectedly.

How can you realise an extensible processing infrastructure which minimises the risk of introducing hidden interdependencies between framework extensions?

To avoid unwanted interdependencies between extensions, it is recommended to decouple the add-on behaviour introduced by two extensions. To achieve this, you must balance two forces: the placement and the centralisation of the control and data flow in your processing infrastructure. Decoupling is best achieved through a decentralised specification of the control and data flow layout, i.e. each extension remains unaware of other, currently active ones. Recording and storing control flow information should also be kept within an extension's realm (i.e., by an intrinsic placement of information).

Following from this, an extension is inserted without full knowledge of the global configuration. Extension developers will not have to be concerned with the global state of the processing infrastructure, the resulting processing behaviour can only be stated (and verified) upon runtime. In other words, you must design your processing infrastructure according to a strict LAYERS structure.

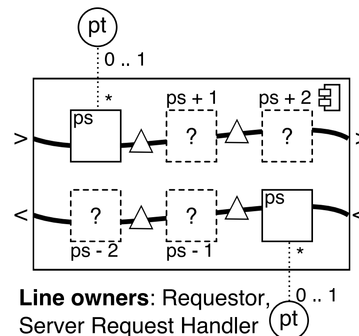


Figure 9: Single-task assignments, variable, and decentralised station layout

Therefore:

Arrange the INVOCATION ASSEMBLY LINES with a variable number of processing stations which chain themselves in a decentralised manner, i.e., through forward references owned locally by each processing station. Attach a single processing task to each station. RECONFIGURABLE PROCESSING PATHS are realised by varying the number and positioning of the processing stations alone.

In some detail: You can learn the essentials of this INVOCATION ASSEMBLY LINE variant from Figure 9 which shows two INVOCATION ASSEMBLY LINES. While they are made of a number of processing stations, each being assigned a single processing task, their exact number and quality is not known to any central entity which is extrinsic to the INVOCATION ASSEMBLY LINES themselves. Most importantly, the line owners do not manage, nor are they aware of the station layout to be enacted when processing an invocation. Rather, each processing station (`ps`) points to a successor station (`ps + 1`), if available. In turn, inserting processing stations on demand requires identifying the right location in terms of its direct antecedent station as the registrar. Therefore, the control and data flow is the result of a decentralised composition process.

This has important consequences, especially for realising PROCESSING SHORTCUTS (pp. 20) and RECONFIGURABLE PROCESSING PATHS (pp. 18). Shortcuts can basically be achieved by resolving the targeted processing step by following the forward references along the station chain. However, the target station must be known to the extension developer and, thus, the source processing station. Also reconfigurations of the station chain happen under conditions of decentralised flow control. It is possible to allow processing station may insert or remove subsequent ones by tracking the forward references. Note, however, that the absolute positioning of a processing stations is not guaranteed in the presence of multiple, active framework extensions which manipulate this station layout.

This SINGLE-TASK PROCESSING STATION strategy, incarnating a LAYER variant for the processing infrastructure, is found for Mono .NET Remoting (Mono/R, see [40, 50]) and Mono Olive (Mono/O; see [41]). Details are given in Section 7 on known uses.

Multi-Task Processing Stations

You are in a situation in which invocation pattern variants (e.g., an in-only MEP [17, 25]) are negotiated through INTERFACE DESCRIPTIONS or, even more lazily, through the INVOCATION CONTEXT. Once registered, this invocation pattern is mapped to a configuration to be applied to the processing infrastructure. Such a configuration involves a specific set of INVOCATION ASSEMBLY LINES. Also, you are required to track the state of the underlying MESSAGE exchanges precisely, e.g., in order to verify completion and failure conditions.

The middleware framework must be able to devise arrangements of INVOCATION ASSEMBLY LINES which can be monitored for specified events (i.e., completions, failures, and notifications) and for processing states. At the same time it must not lose its capability of forming PARTIAL PROCESSING PATHS. How can a processing infrastructure, which appears predetermined in terms of processing steps and operations covered, preserve the adaptability still required?

Similar to the SINGLE-TASK PROCESSING STATIONS (pp. 22), you must review the design dimensions of organising the control and data flow in the light of the above requirements: the centralisation and the placement. A centralised organisation of the processing infrastructure fits the monitoring requirement better than a decentralised one. Therefore, the INVOCATION ASSEMBLY LINES should be managed and tracked by a central *controller entity*.

Where to place the flow information (e.g., processing state flags), which is used to monitor and regulate the control and data flow, is more difficult to answer. An intrinsic placement would store this kind of control information with the core elements of the processing infrastructure, i.e. processing stations. In that sense, they would turn stateful. Statefulness, however, limits the reusability of processing paths in the sense of PARTIAL PROCESSING PATHS (pp. 11). An extrinsic placement would have this information bits managed with the invocation data items, e.g., the INVOCATION CONTEXT. This, however, makes it more challenging to provide for the introspection of the processing state from the angle of the controller entity.

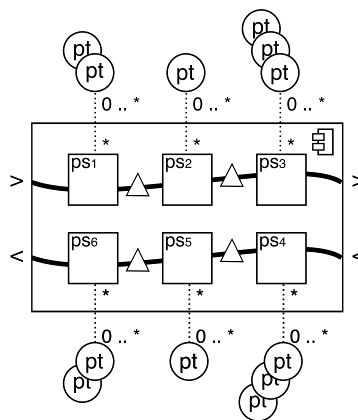


Figure 10: Multi-task assignments, fixed, and centralised station layout

Therefore:
Devise a fixed layout of processing stations which is then shared by different INVOCATION ASSEMBLY LINES. As for fixing, arrange for a manager entity which stores the processing layout as well as the task assignment. Provide for a protocol to access and query the station layout. Make processing stations capable of managing and performing multiple processing tasks. This permits you to enforce dependency constraints between processing tasks without stations being arrangeable.

The solution involves a fixed number of multi-task processing stations: Figure 10 shows two INVO-

CATION ASSEMBLY LINES, each containing three explicitly labelled processing stations (i.e., ps_1,ps_2, etc.). The processing stations are managed by and their participation in realising the two INVOCATION ASSEMBLY LINES is registered with the line owners, either the REQUESTOR or the SERVER REQUEST HANDLER. Thus, the current processing state (in the light of a ruling invocation pattern) can be introspected at any time from the line owners as manager and monitoring entities. The line owner are also responsible to serve with a task registration and introspection interface to client applications. To give the processing infrastructure the needed flexibility, you must allow for fine grained techniques for assigning multiple processing tasks to this set of processing stations. The assignment mechanism needs to be versatile enough to express the precedence constraints as a particular ordering upon assigning tasks. This strategy is often found realised by or deeply integrated with INVOCATION INTERCEPTORS.

Known uses of this INVOCATION ASSEMBLY LINE variant are Apache Axis2/Java (Axis2; see [5, 46, 21]) and Apache CXF (CXF; see [6]). These two frameworks lay out a predetermined, though reconfigurable, arrangement of processing stations referred to as *phases* in both cases. These layouts can only be accessed through and are effectively managed by central manager entities. While these layouts comprise default sets of processing stations, the manager delivers predefined subsets thereof for forming special-purpose INVOCATION ASSEMBLY LINES on demand, e.g., for handling REMOTING ERRORS. In these two variants, the processing stations serve for interception points. So, processing tasks are realised through CXF's and Axis2's INVOCATION INTERCEPTORS. The registration protocol for INVOCATION INTERCEPTORS permits the developer to assign processing tasks in a fine-grained manner, including positioning relative to other INVOCATION INTERCEPTORS and absolute positioning rules. The latter is important to provide guarantees that core processing tasks are executed at the appropriate positions. Details follow in the subsequent section.

6 Motivating Example Resolved

Let us return to the motivating example considered in Section 4.2 and let us briefly walk through applying the small pattern language presented here to structure this design decision space. As framework developers, the example confronts us with the requirement of providing support for optionally securing end-to-end message delivery. This requirement was evaluated against three possible strategies: the refinement of the MARSHALLER or the REQUEST HANDLER components alone, as well as an INVOCATION INTERCEPTOR variant. Each of these approaches reflects requirements and forces captured by the RECONFIGURABLE PROCESSING PATHS pattern (see Section 5, pp. 18). An application which requires delivery encryption should be able to add both encrypting and decrypting operations at processing stages which provide read and write access to the streamed outgoing and the streamed incoming messages. In addition, such an application might act both as client- and server application so that this end-to-end encryption service must be realised for the client- and server roles taken by our middleware framework. Also, this framework extension should only be activated when being used by this application. That is, this application-specific extension should not interfere with other remoting applications integrating our middleware framework. Finally, the extension should be applicable under the entire range of possible service configurations, e.g., different marshaling and transport strategies.

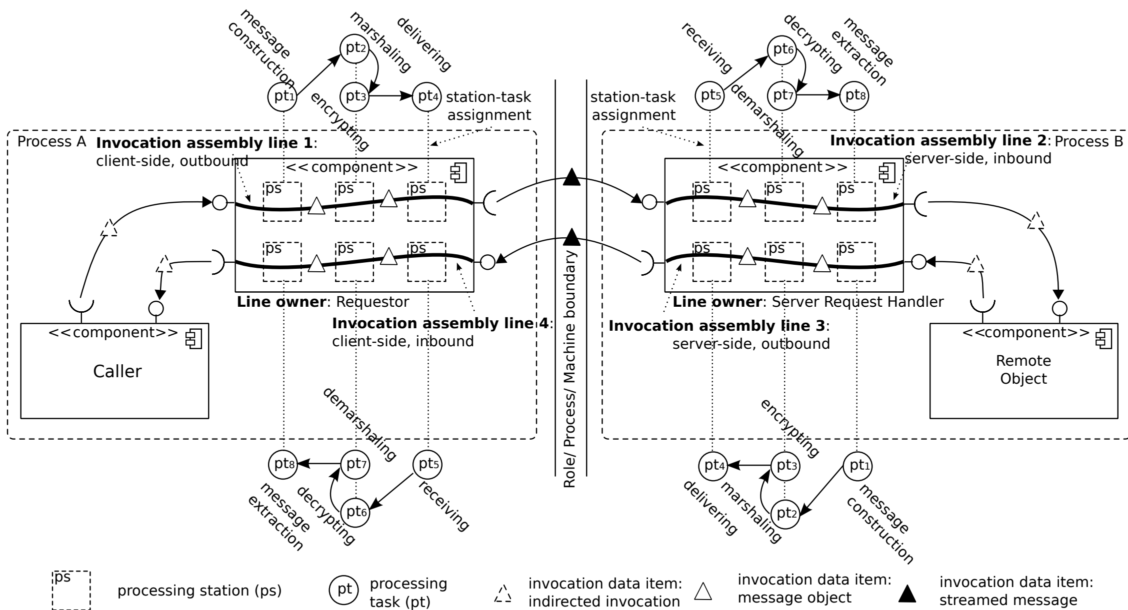


Figure 11: Using INVOCATION ASSEMBLY LINES in a REQUEST-REPLY scenario with message-level encryption

Adopting a RECONFIGURABLE PROCESSING PATHS variant implies applying the INVOCATION ASSEMBLY LINE pattern (see Section 5, pp. 18). Hence, we conceptualise and design the message processing infrastructure in terms of processing stations and processing tasks. Figure 11 provides an exemplary instantiation of the INVOCATION ASSEMBLY LINE pattern. We devise outbound and inbound INVOCATION ASSEMBLY LINES for both the client- and server-side processing infrastructure of our framework. Each INVOCATION ASSEMBLY LINE consists of three processing stations, reflecting the elementary life-cycle stages of the invocation requests and replies processed, such as `objectified`, `marshaled`, and `delivered` (or, `received`). As for the basic processing tasks in a secured REQUEST-REPLY invocation, as shown in Figure 11, both the client-side, outbound and the server-side, outbound INVOCATION ASSEMBLY LINES perform a set of four processing tasks: message construction (pt_1), `marshaling` (pt_2), `encrypting` (pt_3), and `delivering` (pt_4). Similarly, the client-side, inbound and the server-side, inbound INVOCATION ASSEMBLY LINES are characterised by the succession of `receiving` (pt_5), `decrypting` (pt_6), `demarshaling` (pt_7), and `extracting`

(pt_8) invocation data from incoming MESSAGES. This sharing of processing tasks between the client and server side, though in different configurations, reflects a certain processing role symmetry This symmetry mates with the ideas of freestanding MARSHALLER and PROTOCOL PLUG-IN [63, 53] components which can be reused to realise either framework role.

In a next step, we plan to support secure delivery under invocation patterns other than REQUEST-REPLY. Also, secure delivery requires a centralised monitoring of the ongoing processing operations. Hence, we apply a strategy of MULTI-TASK PROCESSING STATIONS (see Section 5, pp. 24). The second processing stations in each of the four INVOCATION ASSEMBLY LINES is responsible for performing two tasks. The sequencing within the two resulting pairs of tasks (i.e., `marshaling`/ `encrypting` and `decrypting`/ `demarshaling`) realises our motivating example of a `Security provider` (see also Figure 4). Again, the symmetry between the client- and server-side roles is preserved. When applying a MULTI-TASK PROCESSING STATIONS strategy, the processing station layout is not only fixed, but also centrally controlled. That is, the REQUESTOR and SERVER REQUEST HANDLER components as owners of the INVOCATION ASSEMBLY LINES store the station layout, provide a management interface to maintain the station configuration as well as the processing task assignments of the stations, and organise the dispatch upon the processing stations. This strategy is commonly implemented by an INVOCATION INTERCEPTOR variant (see, e.g., [53]). By adopting the MULTI-TASK PROCESSING STATIONS pattern, we obtained a design which facilitates realising PARTIAL PROCESSING PATHS (see Section 5, pp. 18).

7 Known Uses

INVOCATION ASSEMBLY LINES are found in a selection of existing middleware frameworks: OpenORB [58], Mono .NET Remoting (Mono/R; see [40]), Mono Olive (Mono/O, [41]), Apache Axis2/Java (Axis2; see [5, 46, 21]), and Apache CXF (CXF; see [6]). In the following, we reflect on instantiations of member entities (i.e., processing stations, processing tasks, line owners, etc.) and their interactions characteristic for the INVOCATION ASSEMBLY LINE pattern.

7.1 OpenORB

The Java-based CORBA implementation OpenORB realises an INVOCATION ASSEMBLY LINE based on a TEMPLATE CLASS [49, 53] collaboration between *client* and *server managers*, on the hand, and the CORBA-specific COMMAND MESSAGE objects (i.e., `ClientRequest` and `ServerRequest`; see [29, 13]), on the other hand.

OpenORB realises the SINGLE-TASK PROCESSING STATION variant of the INVOCATION ASSEMBLY LINE pattern, characterised by (a) a weak distinction between processing stations and tasks and (b) a rigidly fixed number of processing stations or tasks. **Processing stations** are realised as operation records of the COMMAND MESSAGE object-classes. Each COMMAND MESSAGE stipulates a signature interface with a set of deferred operations, that is, HOOK METHODS to be completed by protocol-specific implementations (e.g., RMI [57], IIOP [45], etc.). These operation implementation take the role of **processing tasks**. Notably, processing tasks pertaining to protocol-specific MARSHALLERS can be assigned this way. The processing flow is laid out by the client and server managers in terms of an abstracted call sequence of HOOK METHODS.

The flexibility of task assignments, however, is limited to the possible method combinations along a hierarchy of object-types compliant to the COMMAND MESSAGE interfaces. The task precedence is bound to the call sequence of deferred operations implemented in the line owners and is, therefore, fixed during runtime. The **invocation data items** actually processed are equally encapsulated by the COMMAND MESSAGE objects. The central `ORB` object-classes act as **line owners**. The major limita-

tions of this implementation variant of SINGLE-TASK PROCESSING STATIONS are also discussed in [53].

7.2 Mono .NET Remoting

In Mono/R as a F/LOSS implementation of Microsoft .NET Remoting, the INVOCATION ASSEMBLY LINE variant takes a dominant position in the overall design. Processing tasks are realised by *sinks* [63, 50]. They incorporate properties of both INVOCATION ASSEMBLY LINES and INVOCATION INTERCEPTORS which appear heavily interwoven.

As for INVOCATION ASSEMBLY LINE, the concepts of **processing station** and **processing tasks** fuse to a large extent. They are embodied as so-called *message*, *formatter*, and *channel sinks*. They are supported by *sink providers* or *sink contributors*. They provide an interface which is used by the **line owner** elements, i.e., *contexts* and *channels*, to establish chains of sinks. Each sink provider can only provide for registering a single sink. Message sinks aim at processing objectified representations of MESSAGES while channel sinks operate on streamed forms of MESSAGES. Therefore, the **invocation data items** targeted are clearly MESSAGES. Message and channel sinks are linked in chains to represent what we identified as instantiations of INVOCATION ASSEMBLY LINE variants. Sinks are organised as forward-linked lists and, hence, represent an intrinsic and dispersed design of SINGLE-TASK PROCESSING STATIONS.

Line ownership and **binding scopes** are interdependent. On the one hand, these are variants of CONFIGURATION GROUPS referred to as *contexts*. On the other hand, they cover the scope of PROTOCOL PLUG-INS known as *channels*. Contexts [50] represent controlled execution domains for remote objects and, therefore, participate in realising CONFIGURATION GROUPS. Contexts allow for attaching context-specific activation, lifecycle management, and extension behaviour to remote objects. These are all realised in terms of message sinks. At the provider side, message sinks act as ultimate invocation dispatchers (i.e., the `StackBuilderSink`) and lifecycle managers (i.e., the `LeaseSink`; see [63]). Channels provide the actual core BROKER functionality to the context-bound REMOTE OBJECTS. Provided that target remote objects are collocated, a special-purpose channel (i.e., the `CrossContextChannel`) offers a shortcut invocation path between local contexts. If machine boundaries need to be passed, channels represent CONFIGURATION GROUPS which install and set up PROTOCOL PLUG-INS for the CLIENT and SERVER REQUEST HANDLERS. For the realm of channels, the processing tasks represent core behaviour described for the BROKER and its essential component patterns: We, roughly, find correspondences of pre-, post-, and marshaling phases for either invocation direction. They are realised over the transition from message over formatter to channel sinks [63, 50].

7.3 Mono Olive

Two INVOCATION ASSEMBLY LINE solution variants are found in Mono/O, a F/LOSS implementation of the Microsoft Windows Communication Foundation (WCF), formerly known as *Indigo*. While the first, centred around the design elements of *channels*, is most visible and relevant for the overall design, the second is a small-scale, yet illustrative example hidden in the internals of Mono/O's INVOKER instantiation.

This first INVOCATION ASSEMBLY LINE occurrence is encountered in the collaboration of *channels* and *channel managers*. This collaboration realises the SINGLE-TASK PROCESSING STATION variant of the pattern. Channels represent **processing stations** that are configured to perform a single-only **task** on the invocation data items processed, i.e., MESSAGE operations. Stock channels that come with Mono/O are limited to performing the role of MARSHALLERS (i.e., `MessageEncoders`) and CLIENT or SERVER REQUEST HANDLERS. Certain channels, such as the REQUESTOR-specific `ClientRuntimeChannel`, are core and non-optional elements of a Mono/O setup. Channels form

INVOCATION ASSEMBLY LINES in terms of a *channel stack* or rather a forward-linked list of channels. Internally, they are referred to as *layered channels* in such a configuration.

The idea of describing processing configurations underlying certain invocation patterns, which is central to the INVOCATION ASSEMBLY LINE pattern, is clearly visible in Mono/O's channels. There are different kinds of channels which express three different invocation patterns in terms of their interfaces; REQUEST-REPLY, FIRE AND FORGET, and a PEER-TO-PEER variant. Depending on the enclosing invocation pattern, channels describe two INVOCATION ASSEMBLY LINES in terms of operation records, one for blocking and one for non-blocking scenarios. At runtime, only one INVOCATION ASSEMBLY LINE configuration manifests effectively.

The role of **line owners** is taken by the channel managers which also realise different kinds CLIENT PROXIES. The **invocation data items** processed are the object representations of MESSAGES. The **binding scope** is described by so-called *bindings*, Mono/O's taste of CONFIGURATION GROUPS. They provide custom channel managers and channels to realise certain remoting and transport protocols. They are enacted either through a programming model or by deployment descriptors.

Internally, the INVOKER incarnation of Mono/O which is referred to as `RequestProcessor` realises a second variant of the INVOCATION ASSEMBLY LINE pattern. Its processing infrastructure is organised as a chain of `ProcessorHandlers` that can be compressed or extended to describe the responsibilities of the INVOKER as a configurable set of processing tasks. Currently, this set describes the tasks of dispatching the invocation upon the servant addressed and triggering the provider-side INVOCATION INTERCEPTORS. This second occurrence of the INVOCATION ASSEMBLY LINE is an example of a MULTI-TASK PROCESSING STATION implementation.

7.4 Apache Axis2

Another implementation variant of the MULTI-TASK PROCESSING STATIONS pattern is found in Axis2. It is built around a strong conceptual discrimination between processing stations and processing tasks. **Processing stations** are referred to as *phases* that are laid out in the global deployment descriptor; therefore, phases are defined for the scope of the entire Axis2 BROKER. More recently, the addition of phases through extension modules, referred to as *dynamic phases*, has been considered. **Processing tasks** are represented by Axis2's INVOCATION INTERCEPTORS, i.e., *handlers*, which are organised as *modules*. Modules and per-module (i.e., dynamic) phases foster the idea of the INVOCATION ASSEMBLY LINE pattern. Processing stations are organised into different kinds of INVOCATION ASSEMBLY LINES which are also specified at deployment time as so-called *flows*. The number of flows is predetermined and restricted. Axis2 distinguishes between flows for inward and outward bound invocation data (i.e., in- and out-flow), and in- and outward bound REMOTING ERRORS (i.e., in- and out-fault flow). Each flow manifests as an ordered set of phases. This represents an example of an extrinsically and centrally organised control flow.

The **data item** processed is the INVOCATION CONTEXT which comes as a composite element in Axis2, including the per-interaction, per-operation, per-service contexts. The INVOCATION ASSEMBLY LINES are attached to the combined REQUESTOR and INVOKER entity in Axis2, i.e., the `AxisEngine`, as the **line owner**. In Axis2, we consider the INVOCATION ASSEMBLY LINE to be found in the inter-workings of flows, phases, modules, and, finally, handlers; once per-module phases are fully supported, the reach of this INVOCATION ASSEMBLY LINE variant will be substantially extended.

7.5 Apache CXF

A further MULTI-TASK PROCESSING STATIONS instantiation comes with Apache CXF. **Processing stations**, again first concepts in terms of *phases*, are solely deployed through PASSIVE REGISTRATION [37] upon initialisation time. Custom phase definitions would have to be provided by injecting dedicated *phase managers*. **Processing tasks** are modelled and implemented as `PhaseInterceptors` which can be assigned to (a) multiple processing stations and (b) each phase can be assigned multiple tasks. When assigning a task set to a processing station, the configuration strategy permits to specify relative ordering for tasks handled by the same processing station.

CXF organises its phases into two major INVOCATION ASSEMBLY LINES reflecting inbound and outbound directions, respectively. Each **line owner**, i.e., the REQUESTOR and INVOKER, has a pair of inbound and outbound lines. This leaves aside lines for handling REMOTING ERRORS. REMOTING ERRORS are processed by two dedicated INVOCATION ASSEMBLY LINES which branch from the main INVOCATION ASSEMBLY LINES. Due to the dependency on phase managers, this INVOCATION ASSEMBLY LINE variant is extrinsically mastered and centrally organised. While the INVOCATION ASSEMBLY LINES are bound to their owners, their task assignments can be scoped in a fine grained manner. As for the **binding scope**, `PhaseInterceptors` are registered for either the global, per-binding, per-service, per-client, or per-endpoint scope. The **invocation data items** processed are MESSAGES only.

8 Discussion

The pattern language for INVOCATION ASSEMBLY LINES assists in (a) identifying processing commonalities for different BROKER roles and in (b) revealing the structural equivalence of central collaborators such as the REQUESTOR and the INVOKER. Processing tasks specific to each role are fully qualified by the representational form of invocation data (i.e., kinds and strategies of marshaling and demarshaling, canonical forms of objectified representation, etc.), their processing directions (inward, outward) and the MESSAGE kinds to process. Therefore, the pattern language suggests a data flow view [8] of the middleware. It aims at describing a middleware design as a series of transformations on invocation-related data. From this angle, we aim at identifying design elements that are responsible for the transformations, the elements actually transformed, and the quality of the transformations applied. The conceptual decomposition into atomic concepts such as processing stations organised in processing lines, tasks, invocation data items, and binding scopes permits to express complementary roles (e.g., consumer/provider, publisher/subscriber/notifier, etc.) based on a shared vocabulary.

When designing and developing a middleware framework, you are often required to support more than one remoting technology based on a shared invocation and MESSAGE processing infrastructure. Web Services (WS; [9, 39, 16, 17]), the Common Object Request Broker Architecture (CORBA; [45]), Java Remote Method Invocation (Java RMI; [57]), Java Messaging Service (JMS; [56]), as well as proprietary and ad hoc styles introduce important idiosyncrasies and specify control and data flow requirements which are potentially conflicting when built on top of such a shared infrastructure. Resulting design forces are addressed by the SERVICE ABSTRACTION LAYER [62, 64, 68] pattern. Such an intermediate infrastructure distinguishes between the BROKER core infrastructure and possibly multiple *frontend channels* which mediate invocations in certain remoting styles and technology families. This shields either side, the BROKER core and the frontend channels, from details orthogonal to their concerns. Each frontend channel is realised by a CONFIGURATION GROUP [63]. CONFIGURATION GROUPS make use of the facilities offered by the INVOCATION ASSEMBLY LINE pattern to adjust the processing infrastructure for their needs and assign their processing tasks, represented by a proprietary set of MARSHALLERS, PROTOCOL PLUG-INS, and INVOCATION INTERCEPTORS, accordingly.

In exemplary detail: You might plan to comply with the CORBA [45] or JAX-WS [15] speci-

fications, beyond the core invocation and MESSAGE handling (e.g., MESSAGE formats). In this, you are expected to implement CORBA-specific *and* JAX-WS-specific INVOCATION INTERCEPTOR semantics, i.e., CORBA's portable interceptors and JAX-WS's handlers. This poses important design problems: Do you foresee two widely independent INVOCATION INTERCEPTOR designs co-existing in your framework? Or, do you plan to provide a common INVOCATION INTERCEPTOR instantiation which is suitable for realising portable interceptors and handlers on top? This design problem is aggravated because CORBA's and JAX-WS's INVOCATION INTERCEPTOR instantiations are situated in predefined *failure recovery schemes*. These recovery models (informally labelled "flow stack model" in the family of CORBA specifications [45, Section 1.6.4.3] or "handler execution model" in JAX-WS [15, Section 9.3.2]) define that special-purpose points of interception are to be enforced (in a particular ordering) once an exceptional condition is sensed. This allows developers to foresee limited recovery or, at least, cleanup tasks. INVOCATION ASSEMBLY LINES help understand and realise such deviating control and data flow designs.

A tentative survey of existing middleware frameworks gives credence to this problem statement related to SERVICE ABSTRACTION LAYERS. For example, both Mono/R [40] and Apache CXF [6] provide CORBA frontend channels (i.e., Mono/R's IIOP *channel* and CXF's CORBA *binding*). As for specification-compliant INVOCATION INTERCEPTORS, CXF does not only provide its framework-specific INVOCATION INTERCEPTOR variant (i.e., phase interceptors), but also JAX-WS handlers.

We want to recapitulate the relationship between INVOCATION ASSEMBLY LINE and INVOCATION INTERCEPTOR, commonly found heavily interwoven in known uses of these two patterns. We may attempt to discriminate between their matters by looking at established kinds of pattern relationships; we limit ourselves to the usage relationship as identified by [43].

- INVOCATION ASSEMBLY LINE *uses* INVOCATION INTERCEPTOR: We aim at providing a SERVICE ABSTRACTION LAYER and devise CONFIGURATION GROUPS based thereupon. A variant of INVOCATION ASSEMBLY LINE uses INVOCATION INTERCEPTORS to enforce the decoupling of processing stations and processing tasks. In a straightforward reading, INVOCATION ASSEMBLY LINES assign a single task to each processing station. However, more complex task precedence structures and the need for organising processing tasks in an atomic and modular manner require processing stations to take responsibilities for several interdependent tasks. This task assignment strategy can be realised by devising each processing station as a point of interception, and, therefore, dispatcher for INVOCATION INTERCEPTORS. Tasks, in turn, are realised as concrete INVOCATION INTERCEPTORS to be registered with a particular processing station. It should be possible to express relative orderings of the task-representing INVOCATION INTERCEPTORS to be able to enforce more complex task precedence constraints. Axis2 and CXF provide occurrences of this relationship.

- INVOCATION INTERCEPTOR *uses* INVOCATION ASSEMBLY LINE: A realisation of the INVOCATION INTERCEPTOR patterns calls for a versatile or extensible set of points of interception. As prominently stated in [52, p. 137], designing an adequate model domain of interception points which anticipates the manifold requirements of integrating applications and framework extensions is non-trivial. There is both the risk of devising too limited or too bloated a number of interception points. The INVOCATION ASSEMBLY LINE pattern can help balance these forces by allowing to vary the number of interception points (i.e., processing stations) as needed. This can be used to organise the registration and dispatch of INVOCATION INTERCEPTORS in a more flexible manner.

  Known uses of both the INVOCATION ASSEMBLY LINE and INVOCATION INTERCEPTOR patterns exemplify this use relationship, e.g., *dynamic message sinks* in Mono/R [40] and *message inspectors* in Mono/O [41]. The pattern story on Axis2 in [63, pp. 238] provides an outlook on shifting the responsibility for laying out the model of interception points into the INVOCATION INTERCEPTORS themselves by means of an `AspectHandler`.

You may think of these two patterns as being at either end of a continuous spectrum. In its strictest variations, the INVOCATION INTERCEPTOR pattern has been documented as a hooking technique that preserves orthogonality to the surrounding invocation infrastructure of the BROKER [63, p. 130]. Add-on functionality provided by INVOCATION INTERCEPTOR is allowed constrained access to the overall BROKER state only. This characterisation applies to CORBA's *portable interceptors* and certain uses of INVOCATION INTERCEPTOR in Mono/R, i.e., *dynamic message sinks*. The more this orthogonality and sanity requirements are loosened, the more appropriate is the INVOCATION INTERCEPTOR as a solution part of the INVOCATION ASSEMBLY LINE pattern.

The inter-workings between the INVOCATION ASSEMBLY LINE and INVOCATION CONTEXT patterns must be considered thoroughly. Both as an argument-passing strategy and as a participant in the INVOCATION INTERCEPTOR [31, 65], the INVOCATION CONTEXT already serves the purpose of capturing processing information. By representing an invocation's compositional layout in the INVOCATION CONTEXT, e.g., by storing MESSAGES in different processing states, the necessary processing tasks may be inferred from the compositional state of the INVOCATION CONTEXT. The INVOCATION CONTEXT's composition as a state compound allows for varying processing behaviour.

Alternatively, the INVOCATION CONTEXT can be used to realise a FLAGS FOR STATES [26] strategy. If the processing model throughout the BROKER or parts thereof, i.e., the INVOCATION INTERCEPTOR infrastructure, is organised around state flags, the INVOCATION CONTEXT offers itself as a delivery vehicle. The state flags are accessed at important decision points and used to realise conditional branching. The use of the INVOCATION CONTEXT with state flags is an example of a central and extrinsic organisation of processing behaviour. Considering the INVOCATION CONTEXT as a state compound represents a dispersed and extrinsic approach.

Applying the INVOCATION ASSEMBLY LINE pattern comes with certain advantages and drawbacks. Major advantages result from an increased level of separating between concerns of developing a framework as an intentionally incomplete, cross-domain infrastructure and concerns pertaining to domain-specific framework extensions. Drawbacks can be explained by the increased design complexity, caused by adding a further piece of abstracted-general design to your framework. Besides, we may encounter negative effects on runtime qualities due to more extensive resource usage.

By specifying custom configurations for processing stations, tasks, task assignments, and layouts for the so-realised INVOCATION ASSEMBLY LINES, developers of framework extensions can achieve necessary behavioural variations through a piece of abstracted-general design and a dedicated programming model. This facilitates adding support for component interaction styles and invocation patterns not anticipated initially. The division of development labour, in particular between framework and extension developers, can be deepened.

The ability to adjust the processing infrastructure and the decoupling of processing stations from tasks permits to reuse existing processing facilities for developing new CONFIGURATION GROUPS. Reuse candidates are MARSHALLERS and INVOCATION INTERCEPTORS bundled with existing ones. More generally, INVOCATION ASSEMBLY LINES allow to capture symmetry and asymmetry found crosscutting the BROKER pattern compound decomposed into responsibility-based LAYERS [8]. Components residing at each layer exhibit similar or comparable requirements on the processing infrastructure. INVOCATION ASSEMBLY LINES can be used to express these commonalities and refine variations as a composition and reconfiguration problem. Framework reuse is therefore fostered by an improved composability.

There are important tensions caused by potential gains and losses in modular comprehensibility. On the one hand, actual responsibilities for processing invocation data are more clearly separated from organising them in a particular flow of processing tasks. This permits to reason about MARSHALLER, PROTOCOL PLUG-INS, etc. in a modular manner. In addition, otherwise squattered decision points which shape the control and data flow in the processing infrastructure can be concentrated at certain

places of control. At the same time, the modular self-sufficiency of CONFIGURATION GROUPS is potentially reduced. The INVOCATION ASSEMBLY LINE pattern promotes the idea of expressing a variety of processing task dependencies based on a common infrastructure of processing stations and explicit behavioural models. The expressible variety risks burdening the development of framework extensions with the extra need for scrutinising the subtle details of specifying INVOCATION ASSEMBLY LINES.

Acknowledgments

We would like to thank our EuroPLoP 2009 shepherd Didi Schütz for his constructive and insightful feedback on this paper. Thanks are also due to the participants of the writer's workshop F for providing substantial feedback: Anjali Das, Veli-Pekka Eloranta, Arto Juhola, Farah Lakhani, Marko Leppänen, Ville Reijonen, Martin Wagner, and Tim Wellhausen.

References

[1] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential Serialization for Optimized SOAP Performance. In *Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC)*, pages 55–64, Honululu, Hawaii, June 2004.

[2] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. On the Notion of Coupling in Communication Middleware. In Meersman et al. [38], pages 1015–1033.

[3] N. Allen. Asymmetry Between Listeners and Factories. Discussion Blog [Internet], [unknow location] : Nicholas Allen, 2006 Oct - [cited 2009 June 3], Available from: `http://blogs.msdn.com/drnick/archive/2006/10/25/asymmetry-between-listeners-and-factories.aspx`, 2006.

[4] D. Andresen, D. Sexton, K. Devaram, and V. P. Ranganath. LYE: A High-Performance Caching SOAP Implementation,. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 143–150, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[5] Apache Foundation. Apache Axis2/Java - Next Generation Web Services. `http://ws.apache.org/axis2/`, last accessed: October 13, 2008.

[6] Apache Foundation. Apache CXF: An Open Source Service Framework. `http://cxf.apache.org/`, last accessed: October 13, 2008.

[7] P. Avgeriou. Run-time Reconfiguration of Service-Centric Systems. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 2006)*, Irsee, Germany, 2005.

[8] P. Avgeriou and U. Zdun. Architectural Patterns Revisited – A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, pages 1 – 39, Irsee, Germany, July 2005.

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thattle, and D. Winer. Simpe Object Access Protocol (SOAP) 1.1. W3C Note, W3C, 2000.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, World Wide Web Consortium (W3C), 2008.

[11] P. A. Buhler, C. Starr, W. H. Schroder, and J. Vidal. Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation. In *Proceedings of the 4th International Conference on Web Engineering (ICWE 2004), Munich, Germany*, volume 3140 of *Lecture Notes in Computer Science*, pages 603–604. Springer Berlin / Heidelberg, July 2004.

[12] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture — A Pattern Language for Distributed Computing*, volume 4 of *Wiley Series in Software Design Patterns*. John Wiley & Sons Ltd., New York, 2007.

[13] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture – On Patterns and Pattern Languages*. Wiley Series on Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, April 2007.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, editors. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd., Chichester, England, 2000.

[15] R. Chinnici, M. Hadley, and R. Mordani. The Java API for XML Web Services (JAX-WS) 2.0. Java Specification Request 224, Sun Microsystems Inc., 2005.

[16] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, W3C, 2001.

[17] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 2.0. W3C Recommendation, W3C, 2007.

[18] J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, USA, 1st edition, 1995.

[19] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and Ü. Yalçinalp. Web Services Reliable Messaging (WS-ReliableMessaging) 1.2. OASIS Standard Specification, OASIS Web Services Reliable Exchange (WS-RX) TC, 2008.

[20] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Request for Comments (RFC) 2246, The Internet Society – Network Working Group, 1999.

[21] J. Ekanayake and D. Gannon. Common Architecture for Functional Extensions on Top of Apache Axis 2. Draft report Y790, Department of Computer Science, School of Informatics, Indiana University, 2006.

[22] O. Evans. *The Young Mill-Wright and Millers Guide: Illustrated by Twenty-Eight Descriptive Plates and a Description of an Improved Merchant Flour Mill with Engravings by C. and O. Eveans, Engineers*. Carey, Lea and Blanchard, Philadelphia, 1834.

[23] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments (RFC) 2616, The Internet Society – Network Working Group, June 1999.

[24] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet draft, Netscape Communications, November 1996.

[25] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Simpe Object Access Protocol (SOAP) 1.2: Adjuncts. W3C Recommendation, W3C, April 2007.

[26] K. Henney. Methods for States. In P. Hruby and K. E. Sørensen, editors, *Proceedings of the First Nordic Conference of Pattern Languages of Programs (VikingPLoP 2002)*, Copenhagen, Denmark, September 2003.

[27] M. Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, May-June:66–75, 2004.

[28] M. Henning and M. Spruiell. Distributed Programming with Ice. Manual 3.3.1, ZeroC, Inc., 2009.

[29] G. Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2nd edition, 2004.

[30] D. Jayasinghe. Dynamic Phase support. Mailing List Document, last accessed August 18, 2008, May 2007.

[31] A. Kelly. Encapsulated Execution Context. In *Proceedings of EuroPLoP 2003, Workshop D*, 2003.

[32] M. Kircher, M. Völter, K. Jank, C. Schwanninger, and M. Stal. Broker Revisited. In *Proceedings of EuroPLoP 2004*, Irsee, Germany, 2004.

[33] P. Leitner, F. Reisenberg, and S. Dustdar. DAIOS – Efficient Dynamic Web Service Invocation. Technical Report TUV-1841-2007-01, Distributed Systems Group, Information Systems Institute, Technical University of Vienna, 2007.

[34] C. I. V. Lopes. D: A Language Framework For Distributed Programming. Phd thesis, College of Computer Science, Northeastern University, November 1997.

[35] O. L. Madsen. Towards Integration of State Machines and Object-Oriented Languages. In TOOLS Europe [60], pages 261–274.

[36] D. Manolescu, M. Völter, and J. Noble, editors. *Pattern Languages of Program Design 5*. Addison-Wesley Professional, 2005.

[37] K. Marquardt. Patterns for Plug-ins. In Manolescu et al. [36].

[38] R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Ö. Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, and S. Spaccapietra, editors. *Proceedings of the On the Move to Meaningful Internet Systems Conferences (Part I): CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005*, volume 3760 of *Lecture Notes in Computer Science*. Springer, 2005.

[39] N. Mitra and Y. Lafon. Simpe Object Access Protocol (SOAP) 1.2. W3C Recommendation, W3C, 2007.

[40] Mono Project. Mono .NET Remoting. `http://www.mono-project.com/`, last accessed: February 12, 2009.

[41] Mono Project. Mono Olive. `http://www.mono-project.com/Olive`, last accessed: February 12, 2009.

[42] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web Services Security Core Specification 1.1 (WS-Security 2004). OASIS Standard Specification, Organization for the Advancement of Structured Information Standards (OASIS), 2006.

[43] J. Noble. Classifying Relationships between Object-Oriented Design Patterns. In *Proceedings of the Australian Software Engineering Conference 1998 (ASWEC'98)*, pages 98–109, Adelaide, Australia, November 1998.

[44] S. Y. Nof, W. E. Wilhelm, and H.-J. Warnecke. *Industrial Assembly*. Chapman & Hall, 1st edition, January 1997.

[45] OMG. Common Object Request Broker Architecture (CORBA). Core Specification 3.0.3, Object Management Group, Inc., 2004.

[46] S. Perera, C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. Axis2, Middleware for Next Generation Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 833–840, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[47] J. Postel. Transmission Control Protocol. Request for Comments (RFC) 793, Information Sciences Institute, University of Soutern California; Defense Advanced Research Projects Agency (DARPA), 1981.

[48] J. Postel. Simple Mail Transfer Protocol. Request for Comments (RFC) 821, Information Sciences Institute, University of Soutern California, 1982.

[49] W. Pree. *Framework Patterns*. SIGS Books & Multimedia, 1996.

[50] I. Rammer and M. Szpuszta. *Advanced .NET Remoting*. APress Computer Bookshops, 2nd edition, September 2004.

[51] B. Ramsdell. The TLS Protocol Version 1.0. Request for Comments (RFC) 2633, IETF – Network Working Group, 2004.

[52] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, chapter Interceptor, pages 109–141. John Wiley & Sons Ltd.Wiley, Chichester, England, 2000.

[53] J. Siddle. An interactive pattern story about remote object invocation. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLOP'09)*, Chicago, Illinois, USA, 2009.

[54] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Software – Practice and Experience*, 36:711–759, March 2006.

[55] M. Stal. Using Architectural Patterns and Blueprints for Service-Oriented Architecture. *IEEE Software*, 23(2):54–61, March-April 2006.

[56] Sun Microsystems Inc. Java Messaging Service 1.1. Java Community Specification (last accessed: February 13, 2009), Sun Microsystems Inc., 2008.

[57] Sun Microsystems Inc. Java Remote Method Invocation. White Paper (last accessed: April 14, 2008), Sun Microsystems Inc., 2008.

[58] The Community OpenORB Project. OpenORB 1.4.0. `http://openorb.sourceforge.net/`, last accessed: November 9, 2008.

[59] E. Tilevich, W. R. Cook, and Y. Jiao. Explicit Batching for Distributed Objects. Working paper, Department of Computer Sciences, UT Austin, 2009.

[60] TOOLS Europe, editor. *29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe 1999), 7-10 June 1999, Nancy, France*. IEEE Computer Society, 1999.

[61] M. T. Valente and R. Palhares. Collocation optimizations in an aspect-oriented middleware system. *Journal of Systems and Software*, 80(10):1659–1666, 2007.

[62] O. Vogel. Service Abstraction Layer. In *Proceedings of EuroPLoP 2001*, Irsee, Germany, 2001.

[63] M. Völter, M. Kircher, and U. Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Software Design Patterns. John Wiley & Sons Ltd., Chichester, England, 2005.

[64] U. Zdun. Reengineering to the Web: Towards a Reference Architecture. In *Proceedings of Sixth European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 164–176, Budapest, Hungary, March 2002.

[65] U. Zdun. Patterns of Argument Passing. In *Proceedings of the 4th Nordic Conference of Pattern Language of Programs (VikingPLoP2005)*, pages 1 – 25, Otaniemi, Finland, 2005.

[66] U. Zdun. Engineering Loosely Coupled Software Architectures — A Pattern-Based Approach. Habilitation thesis, Vienna University of Economics and Business Administration, Vienna, Austria, January 2006.

[67] U. Zdun. Pattern-Based Design of a Service-Oriented Middleware for Remote Object Federations. *ACM Transactions on Internet Technology*, 8(3):1–38, 2008.

[68] U. Zdun, C. Hentrich, and W. M. P. V. der Aalst. A Survey of Patterns for Service-Oriented Architectures. *International Journal of Internet Protocol Technology*, 1(2):132–143, 2006.

[69] U. Zdun, M. Völter, and M. Kircher. Pattern-Based Design of an Asynchronous Invocation Framework for Web Services. *International Journal of Web Service Research*, 1(3):42–62, 2004.

[70] C. Zhang and H.-A. Jacobsen. Resolving Feature Convolution in Middleware Systems. *SIGPLAN Notices*, 39(10):188–205, 2004.

A Pattern Thumbnails

Pattern	Problem	Solution
SERVICE ABSTRACTION LAYER (see [62, 64, 68])	Your middleware system must allow for providing and consuming remote objects through multiple channels, i.e., remoting technologies and transport protocols. This channel support should be independent from the core invocation handling on remote objects. New channel should be addable on demand.	The SERVICE ABSTRACTION LAYER adds an extra layer which receives and mediates requests originating from different channels. Each channel contains a channel adapter which translates back and forth requests between the backend and frontend channel formats. This permits you to separate between a core BROKER and frontend channel extensions on top of it.
REQUESTOR (see [63, 66, 13, 32])	How can the assembly of invocation data and disassembly of invocation results (e.g., compiling the data needed, marshaling to and demarshaling from message representations) be organised in a centralised manner? Can we avoid re-implementing this functionality repeatedly when targeting an arbitrary number of REMOTE OBJECTS?	REQUESTORS broker invocation data and results between a calling and called REMOTE OBJECT. They construct canonical, objectified representations of invocation data, i.e., MESSAGES. They orchestrate the subsequent processing steps and, thereby, shield its ancestor LAYERS from remoting details.
INVOKER (see [63, 66, 32])	There are recurring and possibly redundant processing tasks to achieve when a MESSAGE arrives at an transport endpoint. This includes demarshaling and disassembling the invocation data, extracting contextual dispatch data etc. These tasks are general for an arbitrary number of targeted REMOTE OBJECTS. Can we avoid redundancy and resulting complexity in MESSAGE processing, invocation setup, and invocation dispatch?	The INVOKER accepts incoming MESSAGES and proceeds by disassembling them. Based on the extracted object references, transmitted by the remote REQUESTOR, the INVOKER may assemble the actual invocation accordingly and dispatch it to the correct REMOTE OBJECT. The invocation result is obtained and returned to the remote REQUESTOR in reverse processing order.
MARSHALLER (see [63, 66, 32]); relates to the MESSAGE pattern and its variants (see e.g. [12, 29])	Invocation and contextual data must be transported in the shape of MESSAGES, i.e., as structured character and, ultimately, as structured byte streams. Remoting technologies differ in their choice for constructing MESSAGES. How can the transformation back and forth between in-memory representations and MESSAGES be organised?	A MARSHALLER is in charge of transforming invocation data, invocation results, and contextual data in either representation. MARSHALLERS reside at both ends of a remote invocation and they must be customisable to support different marshaling strategies and optimisations.
CLIENT and SERVER REQUEST HANDLER (see [63, 66])	Managing the transport of MESSAGES involves redundant and recurring tasks to be achieved by the REQUESTOR/ INVOKER or REMOTE OBJECTS. Transport-level tasks include connection and resource management, as well as the accommodation of application-level and transport-level modes of synchronisation. In addition, transport errors need to be propagated or constraints enforced (e.g., timeouts). How can we dissociate this responsibility from the REQUESTOR/ INVOKER or the REMOTE OBJECT?	The CLIENT and SERVER REQUEST HANDLER are shared by all REQUESTORS and INVOKERS of a BROKER incarnation, respectively. They enclose all transport-level details, the processing of REMOTING ERRORS, and the transport-specific resource management (e.g., concurrency requirements etc.). They are either directly instructed by the REQUESTOR and INVOKER or yield the control upon reception of transport-level events.

continued on next page

continued from previous page

Pattern	Problem	Solution
REQUEST-REPLY [29]	Two applications communicate through an exchange of MESSAGES. Each MESSAGE realises a one-way conversation. What if the sending application requires a reply from the receiver of the initial MESSAGE?	To realise a two-way conversation, exchange pairs of request and reply MESSAGES. Depending on the intended coupling between the sender and receiver, send the reply MESSAGE either via the request's back channel or, alternatively, via its own communication channel.
FIRE AND FORGET [63]	A client application wants to notify a remote object of an event. Neither a result is expected, nor does the delivery have to be guaranteed. A one-way exchange of a single MESSAGE is sufficient.	A FIRE AN FORGET operations is performed by the REQUESTOR without acknowledging the processing or delivery status to the client. The thread of control is yielded to the client immediately.
MESSAGE [29, 12]	How can we provide (semi-) structured exchange formats for streamed invocation data between remoting ends?	Organise invocation (and contextual) data in terms of MESSAGES which annotate the streamed invocation data with certain meta-data, e.g., identifying the streamed data kind, its origin and destination, size. For processing purpose, provide for a uniform reification in object structures.
REMOTING ERROR [63]	Invocation handling by a BROKER contains many sources of failure. Particular types of error conditions are found within the BROKER, such as network and machine failures, unavailability or misconfiguration of remote objects.	Capture and propagate such error conditions as REMOTING ERRORS between the remoting middleware involved. Allow for discriminating between REMOTING ERRORS emanating from different sources such as MESSAGE handling and transmission.
INTERFACE DESCRIPTION [63]	Developers of client and server applications, and their toolkit of proxy and stub code generators, must access the interfaces of remote objects.	Prepare and offer INTERFACE DESCRIPTIONS which describe the interface of remote objects, e.g. in terms of an interface description language. INTERFACE DESCRIPTIONS negotiate operation signatures and REMOTING ERROR types between the remote ends.
CONFIGURATION GROUP [63]	Remote objects often share configuration properties regarding e.g. marshaling techniques and transport protocols. What is an appropriate scope for defining and activating such properties?	Provide CONFIGURATION GROUPS which organise INVOCATION INTERCEPTORS and MARSHALLERS shared by a coherent group of remote objects. Activating such a CONFIGURATION GROUP means enacting the contained marshaling and extension operations.
INVOCATION INTERCEPTOR [63]	Developers of client and server applications often must provide support and add-on functionality on top of remote invocations (e.g., securing remote invocations). These add-ons should be realised transparently without affecting remote invocations directly. Also, these add-ons can be shared between clients and remote objects.	Provide hooks placed along the invocation path. Have INVOCATION INTERCEPTORS operate on the invocation data directly or have them exchange information via an INVOCATION CONTEXT to realise the add-on services. INVOCATION INTERCEPTORS are managed by middleware users to create extensions.
INVOCATION CONTEXT [63]	How to organise and manage contextual or auxiliary data related to underlying remote invocations without modifying the latter and breaking orthogonality?	Contextual invocation data is embodied by a dedicated object structure, the INVOCATION CONTEXT. This object structure provides a uniform interface for manipulating and extracting this context data. In addition, it serves as a CONTEXT OBJECT [65] for arguments passing between e.g. INVOCATION INTERCEPTORS [63].

Table 1: Thumbnail sketches of relevant remoting patterns (see e.g. [63, 32, 66])

B Invocation Data Items

The following four kinds of invocation data items can be found in BROKER-based middleware frameworks:

- A MESSAGE [29, 12] provides means to exchange core, contextual, and auxiliary invocation data between remoting ends, i.e., across machine and process boundaries. MESSAGES allow us to stream in-memory information (e.g., requests and replies) into structured byte and character sequences. By structured, we mean that MESSAGES contain annotating meta-data which describes the streamed data and facilitates its interpretation and restoration into in-memory representations. This meta-data identifies different types of streamed data (e.g., the operation name, parameters, etc.), its size, its origin, and its destination. Examples include a wide range of binary (e.g., CORBA's IIOP binary encoding [45] or ICE's binary encoding [28]) or structured markup MESSAGE formats (e.g., XML [10] encodings such as the different SOAP/XML variants [9, 39]).
- An INTERFACE DESCRIPTION [63] represents the interface of the remote object accessed by the client component. These INTERFACE DESCRIPTIONS are important for client developers. On the one hand, they permit developers to inspect interface capabilities and generate requests manually. On the other hand, INTERFACE DESCRIPTIONS assist in generating client-side code representations of remote interfaces, either ahead of time (i.e., by a code generator) or just in time (i.e., by means of reflection). INTERFACE DESCRIPTIONS are often realised through an interface description language. Commonly known examples are the Web Service Description Language (WSDL 1.1/ 2.0; [16, 17]) and the CORBA Interface Description Language (IDL; [45]).
- INVOCATION CONTEXTS [63] are used in situations that demand the exchange of contextual invocation data which describes certain conditions, constraints, and processing rules imposed upon the underlying remote invocation (for instance, negotiating message exchange patterns as described by WSDL 2.0 [17]). At the same time, it is tedious and invasive to attach this information to the core invocation data explicitly. This would effectively clutter the signature interfaces of remote operations. Therefore, dedicated object structures are used to pass contextual invocation data through and between the client and server endpoints. For transport, INVOCATION CONTEXTS are streamed into dedicated parts of invocation MESSAGES. Consider SOAP headers [9, 39] or CORBA service contexts [45] as prominent examples. Note that INVOCATION CONTEXTS are also important for realising an extension infrastructure for add-on services in your middleware framework. This will be discussed in the next section.
- REMOTING ERRORS [63] are particular exceptional values reported by the REQUESTOR, INVOKER, and the REQUEST HANDLERS upon sensing errors in processing invocations and MESSAGES, as well as in connection and transport management. REMOTING ERRORS are also communicated between client and server endpoints and, thus, appear in the form of particularly structured MESSAGES. They are also commonly mapped to concrete object types in order to be injected into the local exception handling. For example, SOAP *faults* [9, 39] are used in the context of Web Services to signal REMOTING ERRORS in invocations.

C Adaptability Requirements

C.1 Orthogonal Add-on Services

By orthogonal add-on services, we mean extensions which wrap around remote invocations without affecting the structure and format of the core invocation data and its processing. In addition, they may apply to several variants of remote invocations at the same time. Examples include security-related

add-ons (e.g., authentication and authorisation mechanisms), transaction control, logging facilities, persistent storage, and reliable messaging solutions. The example of the `Security provider` and selective encryption (see also Figure 4) falls into the category of orthogonal add-on services. Further examples and elaborations on add-on services are provided in [52] and, in particular, [63].

C.2 Invocation Styles

When creating a BROKER-based middleware, you will find yourself confronted with the requirement to support multiple invocation styles, e.g., kinds of two-way and one-way invocations. This is often demanded by families of remoting technologies. For instance, Web Services (WS) describe different message exchange patterns (MEPs; [39]) between client and server components, such as out-in, in-only, and so on. Invocation styles represent an inherently crosscutting concern, i.e., they affect several LAYERS at once. For instance, different invocation styles require different REQUESTOR and INVOKER interfaces and, therefore, introduce a coupling between the BROKER core and the integrating client and server components. Transport-level requirements often differ substantially and cause special treatment by the CLIENT and SERVER REQUEST HANDLERS.

Invocation styles fall into a set of common invocation patterns [69, 63]. For the scope of this paper, we emphasise the following two invocation patterns:

- The REQUEST-REPLY pattern [29] describes a two-way conversation between a client and a server component. It involves two MESSAGES being exchanged. This pattern is found in component interactions adopting remote procedure calls (RPCs), for instance, but is not limited to these.
- The FIRE AND FORGET invocation pattern [69, 63] captures kinds of one-way conversation between a client and server component. This pattern comforts scenarios in which no invocation result is expected and the reliable delivery *is not* guaranteed on behalf of the client application. There is only a single MESSAGE exchange between the two remote ends. FIRE AND FORGET operations are commonly found in event-based and publish-subscribe component interactions.

Invocation patterns are characterised by different kinds of invocation artefacts, e.g., MESSAGES representing invocations or notifications [29, 12], and their patterns of exchange (e.g., one-way, two-way). The multitude of design requirements and design options is further aggravated because invocation patterns, being located at the application level, map to *communication abstractions* (e.g., send/receive) at the transport protocol level. Invocation patterns are further qualified by different coupling dependencies in terms of location, time, and process synchronisation [2]. We find, for instance, both REQUEST-REPLY invocations that realise a process synchronisation (i.e., blocking REQUEST-REPLY) and those which don't (i.e., non-blocking REQUEST-REPLY). Finally, providing support for different invocation patterns at both the client- and server-side of your framework adds further complexity to your design.

C.3 Bypassing of Layers

Bypassing describes the capability of selectively omitting processing steps (e.g., the processing of MESSAGES by the MARSHALLER) in the LAYERS structure of your middleware framework. This serves the purpose of realising optimisations and gaining a certain flexibility. Looking at the selective encryption example (see also Figure 4), once the `Security provider` has been successfully added to the processing flow of your middleware framework, you still want to preserve the flexibility of disabling the encryption on selected remote invocations or certain endpoints. More generally speaking, areas of bypassing include:

- Stub- or proxyless invocation handling [11, 33]: Using negotiated INTERFACE DESCRIPTIONS to perform remote invocations often means to couple client and server components to rigid signature interfaces. Upon interface changes, client and server components must track these changes (e.g., by regenerating stub or proxy code). This adds maintenance overhead and means a form of coupling to be avoided in certain component interaction styles such as event-based interactions. Therefore, client- and server-side interface proxies are bypassed so that REQUESTOR and INVOKER are directly addressed to handle and to dispatch invocations.

- Message caching [4], partial (differential) marshaling [1]: These examples try to avoid the repetitive streaming into and the redundant objectifying from MESSAGES. This is achieved by factoring out MESSAGE parts (e.g., XML markup) which remain unchanged between invocations. In other words, these strategies mean to bypass the MARSHALLER, at least partially.

- Batched invocations [59]: Remote invocations are often performed in an atomic manner and map directly to invocations upon single operations of remote objects. These are often small-scale operations so that the tasks performed by the client components are split into a series of successive remote invocations on the same remote object. Given that remote invocations are comparably expensive to process, negative impact on the overall client-perceived performance can be expected. Grouping invocations into batches and processing them in such lots is a viable option, provided that batched invocation results can be assigned to the individual invocation sources. Besides, you have to treat intermediate invocation results specifically. Batching, therefore, groups processing steps into blocks of request and, then, blocks of reply handling, rather than processing pairs of requests and replies as shown in Figure 4. From the perspective of a member invocation in a given batch, different processing steps (e.g., marshaling and/or transport operations) are effectively omitted. They are only performed on the containing batch once.

- Collocation [61]: In case the targeted remote object resides within the same machine or even process boundaries, important steps of processing the remote invocation can be spared. In particular, certain services offered by the MARSHALLER and REQUEST HANDLERS are superfluous.

C.4 Role Distribution

We have explained how invocation and MESSAGE handling faces extension requirements which cross-cut the LAYERS structure of the BROKER. The LAYERS structure, while highlighting most general aspects of the BROKER, introduces a strong notion of CLIENT-SERVER [8] relations, not only between the higher and the lower-level LAYERS, but also between a client application and a remote object. We say that the distribution of the client and server roles over applications and components is *asymmetric*. As a result, there are disjoint sets of client- and server-only components. Asymmetry means that (possibly multiple) client-only components are meant to initiate requests for remote invocations from server-only components, or, more precisely, from their remote objects exposed. This asymmetry between client-only and server-only components manifests in the unequal distribution of lookup information, i.e., server applications are only aware of clients for the scope of an incoming invocation. Also, different resource management strategies apply to client-only and server-only applications, the latter putting emphasis on handling multiple and possibly concurrent invocation requests. While the CLIENT-SERVER conceptualisation is simple yet powerful for capturing remote invocation details in distributed object systems, it is the exact opposite of what is found in certain middleware framework designs and of what is required in advanced deployment scenarios. The following examples provide evidence for different appearances of *role symmetry*. In these examples, an application takes and its underlying middleware framework supports the role of both server and client within single component interactions.

- Layer symmetry in invocation and MESSAGE processing: The functional decomposition of the BROKER into LAYERS itself reflects "that there is a certain symmetry between client and server

in the layered architecture. This is because the processing patterns at each layer depend on their remote counterpart" [63, p. 128]. This *layer symmetry* [52, p. 136] is yielded by a dependence between remote components within a given LAYER. This dependence is due to most processing steps being performed on invocation data at either the client or server side requiring a counter operation at the respective remote end. Applied to the exemplary BROKER shown in Figure 3, this becomes most visible for the MARSHALLER. Each marshaling operation demands a corresponding demarshaling operation, and vice versa. The MARSHALLER components can, thus, be potentially shared in the client- and server-specific infrastructures. This equally applies to processing steps for add-on services. In our motivating example of the `Security provider` (see Section 4.2), en- and decrypting operations also form pairs across the remote ends. You can translate this symmetry into a design of entities and processing behaviour shared by client- and server-specific invocation handling.

- Asynchronous REQUEST-REPLY invocations through RESULT CALLBACKS: You will find situations in which the interacting client and server applications do not need to be coupled through process synchronisation, yet a reply is expected by the client application. That is, the processing infrastructure of the middleware yields the thread of control to the client application after having processed the invocation request. Conversely, at the server side, the server application yields the thread of control to the processing infrastructure after having received the invocation dispatch. Now, one strategy available to communicate a result back to the originating client and, thus, completing the REQUEST-REPLY exchange, are RESULT CALLBACKS [63, 8]. Following this pattern, the server side actively notifies and delivers the invocation result to the client end. At the client side, this is made possible by a RESULT CALLBACK object. This represents a special-purpose remote object exposing a callback interface. Upon reception of a callback invocation, the callback object processes the result in accordance with the client application logic. As the callback object is not only part of the client application, but also acts as a remote object, a particular role symmetry can be stated. Consequently, your middleware framework needs to provide server-specific facilities to client applications under the conditions of asynchronous REQUEST-REPLY invocations.

- Service-oriented, adaptive system distribution: The requirements of adaptable configurations in distributed, service-centric systems [7] are another argument which helps explain the importance of symmetric role distributions for middleware designs. By changing configurations, we mean that, on the one hand, service consumers and providers become available or turn unavailable dynamically (at runtime) and that, on the other hand, the composition of service providers must remain adaptable at any time to deliver guaranteed quality attributes. An important example are distributed systems based on workflow engines which integrate with third-party and legacy systems through service adapters and interfaces (see e.g. [67]). In these cases, certain requirements are imposed on your middleware framework. First, if it runs the workflow engine on top, it must support both the client- and server-side of handling remote invocations. In other words, the workflow engine also offers a service interface to receive task results in a process-decoupled manner. In turn, second, the service-providing applications must be in the position to both receive invocation requests (as servers) and deliver corresponding results in an asynchronous manner (as clients). Third, the decentralised addition, removal, and upgrade of service-providing components requires not only facilitating the creation of service adapters and interfaces, but also the propagation of interface changes (e.g., dynamic invocation). This is, for instance, where LAYERS bypassing in your middleware comes into play.

  Leela [67] realises such a service-oriented middleware framework. It is built around the idea of peer components which participate both in client and service applications. That is, each peer acts both as a remote object and an invocation client. Thus, the peer is in the position to connect to REQUESTORS for handling their invocation requests as well as to serve invocation dispatches from INVOKERS. Beyond this symmetry in invocation roles, peers are organised in terms of federa-

tions which provide lookup services to all peers under equal terms, so loosening the asymmetry in distributing lookup information (e.g., object references).

There are many ways to make use of these forms of role symmetry in your framework design. One example we experienced is the integration of major components of REQUESTOR and INVOKER incarnations into single framework entities. In Axis2 [5], the `AxisEngine` object-class captures essentials of the INVOKER and the REQUESTOR behaviour in its `send` and `receive` operations. The `AxisEngine` is used by the SERVER REQUEST HANDLERS, i.e., the transport listeners in Axis2, to dispatch incoming invocations and, conversely, by `ServiceClients` as a part of Axis2's client-side proxies to process outgoing invocation requests. Another example is found in the Windows Communication Foundation (WCF; see e.g. [41]). Beyond INVOKER and REQUESTOR (i.e., the *channel factory* and the *channel listener*, respectively), also the REQUEST HANDLER instantiation (i.e., the *channels*) is shared between the server- and client-specific infrastructures (see e.g. [3]). This uniform design is achieved by the WCF channel factory and the channel listener deriving from a shared *channel manager* entity.

Note, however, that under certain conditions it may be advisable to break this symmetry in framework design intentionally. To name an example, certain lifecycling requirements and, thus, resource management strategies may vary between the server- and client-specific elements of the processing infrastructure. To provide scalability at the server side, imagine that you realise a pooling of INVOKERS for scaling out the handling of concurrent invocations. This pooling, however, is not necessarily needed at the client side (see e.g. [3]). Therefore, expect a coexistence of symmetric and asymmetric design elements between the client- and server-side infrastructure of your middleware framework.