Conceptual Architecture Patterns

Technische Berichte
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

2

Conceptual Architecture Patterns: FMC-based Representations

Bernhard Gröne and Frank Keller
(eds.)

April 2004

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber:	Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam
Redaktion	Bernhard Gröne und Frank Keller
EMail	{bernhard.groene, frank.keller}@hpi.uni-potsdam.de
Vertrieb:	Universitätsverlag Potsdam Postfach 60 15 53 14415 Potsdam Fon +49 (0) 331 977 4517 Fax +49 (0) 331 977 4625 e-mail: ubpub@rz.uni-potsdam.de http://info.ub.uni-potsdam.de/verlag.htm
Druck	allprintmedia gmbH Blomberger Weg 6a 13437 Berlin email: info@allprint-media.de

Heft 2 (2004)
ISBN 3-935024-98-3
ISSN 1613-5652

Conceptual Architecture Patterns: FMC–based Representations

Bernhard Gröne and Frank Keller (editors)
Research assistants of the chair "Modeling of Software–intensive Systems"
Hasso–Plattner–Institute for Software Systems Engineering
P.O. Box 900460, 14440 Potsdam, Germany
E-mail: `{bernhard.groene, frank.keller}@hpi.uni-potsdam.de`

Abstract

This document presents the results of the seminar "Conceptual Architecture Patterns" of the winter term 2002 in the Hasso–Plattner–Institute. It is a compilation of the student's elaborations dealing with some conceptual architecture patterns which can be found in literature. One important focus laid on the runtime structures and the presentation of the patterns.

Contents

1. Introduction

1.1. The Seminar

In the winter term of 2002, we offered a seminar "Conceptual Architecture Patterns" for students in the 5th semester of Software Systems Engineering. They should learn what literature says about architecture of software–intensive systems and combine the conceptual architecture view with architecture patterns. The seminar was divided into two parts: In the first part, the students had to read and present the different architecture views of [HNS00] and learn about patterns presented in [GHJV94]. In the second part, they used this knowledge to read the POSA books [BMR$^+$96, SSRB00] and present the architectural patterns of the books with the conceptual architecture view in mind.

Another important topic was to find an adequate notation for the pattern and to discuss the alternatives, like the Fundamental Modeling Concepts (FMC) [KTG+02][KW03]. As the students had examined the Apache HTTP server very closely [GKK03], it was obvious to look for applications of the patterns in Apache or similar systems.

1.2. Literature

The patterns in this compilation have been taken from the books "Pattern–Oriented Software Architecture" Volumes 1 and 2 [BMR+96, SSRB00]. The conceptual architecture is one of four views introduced by Kruchten [Kru95] and refined by Hofmeister, Nord and Soni in their book "Applied Software Architecture" [HNS00]. To introduce the students to patterns, they had to present some design patterns of the Book "Design Patterns" by Gamma, Helm, Johnson and Vlissides [GHJV94].

References

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*, volume 1. Wiley, 1996.

[GHJV94] Erich Gamma, Richard Helm, Raph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[GKK03] Bernhard Gröne, Andreas Knöpfel, and Rudolf Kugel. The apache modelling project. Web site, 2003. apache.hpi.uni-potsdam.de.

[HNS00] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. The Addison-Wesley object technology series. Addison Wesley Longman, 2000.

[Kru95] P. Kruchten. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, 11 1995.

[KTG+02] Frank Keller, Peter Tabeling, Bernhard Gröne, Andreas Knöpfel, Oliver Schmidt, Rudolf Kugel, and Rémy Apfelbacher. Improving knowledge transfer at the architectural level: Concepts and notations. In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the SERP'02, the international conference on software engineering research and practice, las vegas*, pages 101–107. CSREA Press, June 2002.

[KW03] Frank Keller and Siegfried Wendt. Fmc: An approach towards architecture-centric system development. In *Proceedings of 10th IEEE Symposium and Workshops on Engineering of Computer Based Systems, Huntsville Alabama USA*, 2003.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture — Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

Pipes and Filters Architectural Pattern

André Langhorst, Martin Steinle
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
`{andre.langhorst, martin.steinle}@hpi.uni-potsdam.de`

Abstract

Applications that extensively perform processing of data chunks within various distinguishable processing steps suffer inflexibility in reuse and recombination of processing steps, if interweaved into a monolithic assembly. The Pipes and Filters architectural pattern we present approaches these problems by exploiting the benefits of uniform interconnection methods to construct chains of non-interdependent processing entities. Further we present a graphically simple representation to communicate this architectural pattern effectively.

Keywords: Software architecture, modularization, pipes and filters, architectural patterns, modeling, FMC

1. Introduction

Applications benefit from employing the Pipes and Filters architectural pattern where large amounts of data are to be processed, such as with web servers and rendering, imaging or sound processing as well as applications of message processing, such as applications of enterprise application integration (EAI), business process engines and processing engines for stackable protocols, for example web services stacks. First we show problems inherent to a monolithic design of processing steps. In section 2 we present the Pipes and Filters patterns as a solution addressing these problems. A discussion of advantages and disadvantages of employing the pattern follows in section 3. Effective communication is best supported by using graphical notations, accordingly we discuss graphical notations for communication means in section 4, whereas sections 5 and 6 cover extensions of the basic pattern and a reference implementation of the pattern respectively.

1.1. Application

As outlined in the section before large amounts of either chunkable data or single messages are to be processed. Chunkable means that data chunks belonging to a larger set of data may be processed in any order; in effect that is no interdependencies between data chunks that dictate the order in which individual chunks are to be processed. If single data units are not associated with a larger set of data, as it is the case with messages, the data implicitly is chunkable. Naturally a processing order too implicitly is obeyed by the sequence of processing steps itself.

The Pipes and Filters pattern is applicable, if repeatedly numerous individual, distinguishable processing steps, that perform processing and transformation of data, are involved within a sequence of processing steps.

1.2. Problems

Let the processing of the examples given before be assembled – this is in software, hardware or both – into a monolithic component.

The outcome is an inflexible structure; no easy reuse, recombination, adding or omitting of processing steps is possible anymore.

For example, let there be two complementary processing components I and O that transform a stream of data in some way, one requirement initially was that the data is encrypted before it leaves component O and accordingly is decrypted on entering component I. Henceforth, if such a processing component needs to be reused with changed requirements, either resources may be wasted or the components may be rendered worthless.

Given lessened requirements now do not include encryption and decryption anymore, then if simply a transportation layer is used between component I and component O simply resources – for example, cryptographic hardware devices, or processing power – are wasted. If encryption is performed in software for example and CPU power is a precious resource then this may turn out to be a problem.

Given that the output of component O needs to be intercepted before handing it into component I, encryption may render this impossible and thus render the components worthless. The same is true if component I should accept non-encrypted or differently encrypted input. Generalized, if steps need to be modified to large extents or need to be replaced completely the effort required is huge resulting in an increased development effort.

This is especially true if for example the component created consists jointly of hardware and software and a move to a different platform is required: Some hardware components may not be available anymore, not work flawlessly or not work at all within the changed environment, the programming language may have changed and so on.

2. Solution: Pipes and Filters

Processing steps are connected by pipes to create filter chains. A sequence of generally independent, but adjacent, processing steps performed by filters that are connected by uniform channels, labeled pipes, used to incrementally process data constitutes a filter chain.

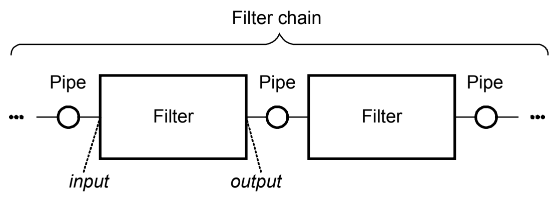


Figure 2-1 – Basic filter chain with data flow from left to right

2.1. Filter chains

Input to the filter chain is data from another filter or a data source. Output leaving the filter chain is sent to another filter or a data sink. Pipes and filters, perceived as filter chains, constitute the building blocks for the Pipes and Filters pattern.

2.2. Pipes

Pipes do not transform data, generally buffer data and function as the uniform interconnection mechanism connecting at least two filters. Pipes are implemented by for example function calls, OS-pipes, message channels and IPC-channels.

2.3. Filters

Filters do arbitrary processing and transformation, such as data enrichment or refinement. Filters consume data from an input and deliver it to an output.

2.4. Key aspects

The Pipes and Filters pattern uses filter chains to exploit the following key aspects to a certain extent (see section 3).

2.4.1. Uniform communication

Pipes provide uniform communication means between filters.

2.4.2. Incremental processing model

An incremental processing model, utilizing data chunks as smallest entity of communication, underlies the Pipes and Filters pattern.

2.4.3. Stateless filters

Filters generally are stateless, both filters and data chunks are not interdependent.

2.4.4. Endpoints

Support for many different media with respect to input/output for the whole chain is gained through exchanging terminating filters which in turn are connected to data sinks and sources respectively. Examples for data sources/sinks include files, sockets, keyboards/terminals and sensors/actors.

2.5. Classification of filter chains

There are two common classifications that can be applied to filters and filter chains. The behavioral classification helps to spot autonomous active filters and demand-activated passive filters while the functional classification helps to indicate the task the filter chain performs and filters within the filter chain perform. These classifications do not always exactly fit to a certain filter or filter chain respectively.

2.5.1. By behaviour

Active filters actively pull data from and push data to pipes down the pipeline where generally most other filters are passive. Analogous, pulled data flows towards the filter and data pushed down the pipeline flows away from the filter. Active filters may be implemented as a separate process, a separate thread or program or even be represented by a separate component or computer. Pulling means sending a request for data which may be satisfied or propagated within a pull filter chain, once the request is satisfied it is pulled backwards through all filters that propagated the pull-request. Pushing simply means supplying data chunks to the next filter after filter-intrinsic processing has been performed.

Passive filters are activated by receiving pushed data, or by receiving a pull request. If the request for data cannot be satisfied it is propagated and once the filter, the request was propagated to, sends data the filter performs processing and answers the request it received.

Common combinations of active and passive filters include: active filters with passive filter chains attached, active filters with two passive filter chains, one pull and one push pipeline, serving as input and output to the filter respectively. Many more combinations of filters, for example several active filters with several push and pull chains each, are valid.

2.5.2. By function

Input filters and input filter chains are associated with receiving information and corresponding actions. Common tasks of input filters include: Decryption, deserialization, detaching (of attached headers or files for example), decoding, reading values.

Output filters and output filter chains are associated with sending information and corresponding actions. Common tasks of output filters include: Encryption, attaching, encoding, writing values, transforming data.

A strict classification by function may not be accurate if a filter performs hybrid tasks.

2.6. Granularity

The pattern can be applied to very low level tasks, such as stream processing in imaging systems where data on the level of bits and bytes is handled incorporated into a single microchip ranging to very high level tasks, such as messaging systems for business processes where very complex data is passed around within inter-organizational networks.

3. Discussion of advantages and disadvantages

3.1. Benefits

Applying the Pipes and Filters pattern reasonably has specific benefits with regard to architectural attributes of which we list the most important ones. Most of them are founded in the modular approach of the pattern.

3.1.1. Ease of recombination and reuse

A standardized interface both of filter and pipe components allows filter chains to be combined and recombined very easy. By this, many filter chains with different behavior can be produced very fast. For example, UNIX operating systems provide a uniform pipe mechanism and a lot of tools such as "cat" and "sort" which can be combined in any way with a simple shell command.

Also, Pipes and Filters simplifies reuse. New filter chains, even with completely different behavior, can be designed by rearranging filters of an existing filter chain, or adding some new filters. In addition, complete filter chains (without data source and sink) can be used as one filter in a new chain. The gain experienced through replaced endpoints for example include facilitation for supporting multiple file formats, network protocols, databases.

3.1.2. Efficiency by parallel processing

Through incremental processing of data, active filters running on a multiprocessor system or in a network can perform their functions in parallel, because all filters can already start to work on the partial results of its predecessors instead of having to wait for their completion. This can improve the performance of the system using a filter chain.

3.1.3. No intermediate files necessary

Computing results using several programs is possible without pipes, by storing intermediate results in files. This approach disables parallel and incremental processing of data and is errorprone if processing stages have to be set up every time a system is run. Using Pipes and Filters removes the need for intermediate files and enables incremental and parallel processing.

For debugging and testing purposes, you may want to see intermediate results. This can be achieved easily by inserting a T-junction into the pipeline. A T-junction is a special filter that does not modify data, but only writes it to a second destination, for example a file.

3.2. Disadvantages

There are some cases when applying the Pipes and Filter pattern introduces several liabilities of which we again list the most important ones.

3.2.1. Error handling

Error handling is the biggest problem of Pipes and Filters, because pipeline components have only one possibility of communication, the data stream. At least error reporting can be done using a separate output channel, but in most cases the only possibility is to restart the pipeline if an error occurred and to hope that it will complete without failure. If error handling is important, different architectures such as Layers should be considered.

3.3. Sharing state information

If different processing stages must share a large amount of global data, the Pipes and Filters pattern can be inappropriate, because it introduces dependencies between the filters, reducing the possibility to recombine filter chains and to process data in parallel.

3.3.1. Expensive pipe mechanism

If pipes and filters have different data formats, the overhead of converting data can prune the benefits of Pipes and Filters, especially the performance improvements.

The features offered by pipes, such as buffering, queuing and transferring data, allocate resources that limit the number of pipes.

3.3.2. Limited scalability

Due to the linear architecture of Pipes and Filters, the throughput of a filter chain is limited by the throughput of its slowest component and it is not possible to deploy a second instance of this component. So scalability of a filter chain is limited to the scalability of its components.

3.3.3. Small amount of data or low complexity of processing steps

The cost for transferring data between filters can be relatively high compared to the cost of the computation a single filter performs, so the system will chiefly be occupied with data transfer instead of computations. Besides, if the amount of data to process is very small, the cost of building a filter chain can be too high.

4. Notational representation

Graphical representations are well suited for improving knowledge transfer on the architectural level. We present a proposal for modelling Pipes and Filters using FMC. We present detailed models for the most important cases to improve comprehension regarding the Pipes and Filters pattern as well as simplified models to be used as a means of communication.

4.1. Detailed representation

For all examples the actors A and B represent individual filters which are connected through varying mechanisms, the pipe.

4.1.1. Push

What happens when an actor A initiates a push to B through a pipe is depicted in Figure 4-1. The actor sends the data through the data channel to the pipe that buffers all data. Either an overflow is indicated and A is stopped by the pipe or A continues to send and implicitly or explicitly receives ACKs from the pipe. While receiving data the pipe sends data to B and itself can be stopped by receiving a notification of buffer overflow from B. The exact implementation of the pipe for the most part does not matter at an architectural level. Even if we use procedure calls we assume the model depicted in Figure 4-1 holds.

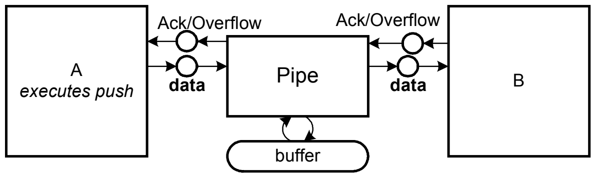


Figure 4-1 Push method in detail

4.1.2. Pull

The pull example depicted in Figure 4-2 is similar to the push example except that requests for data are issued to the pipe which satisfies them from its buffer and sends it similarly, with ACK and overflow, as explained for Figure 4-1. If the request cannot be satisfied it is propagated to filter B, which eventually satisfies the request. For simplicity we ignore the case where the pipe buffer is full with requests and assume that it simply accepts no more requests if the request buffer is full and the requesting party has to resend its request.

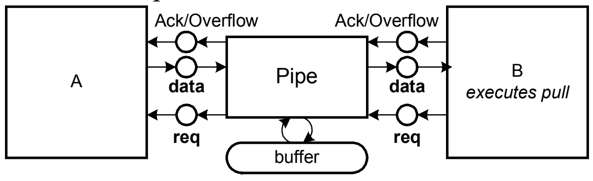


Figure 4-2 Pull method in detail

4.1.3. Two active filters

Two active filters where one filter pushes to and one filter pulls from the pipe is merely a combination of the two cases above.

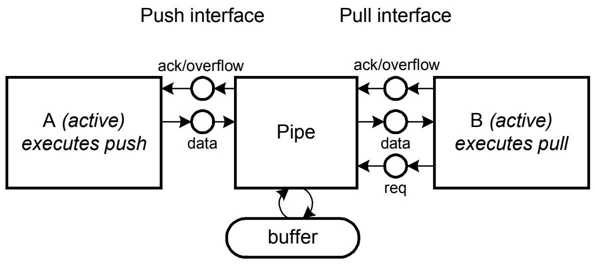


Figure 4-3 Active push filter, connected with active pull filter

4.2. Simplified representation

We reduce unnecessary complexity by abstracting from all error-handling and flow-control and even the pipe as an active element itself. Graphically, we use quasi-textual descriptions to underline data flow, which is crucial to understand how data is passed around in complex compositions. An arrow is drawn within the channel, whose direction matches the direction of data flow. Furthermore we stretch the channels to create a resemblance to pipes and queues graphically. Finally we place "Pipe" within the channel, using "P" as a shorthand for "P" in complex diagrams thereafter. It should be obvious then that the textual description "Pipe" not only names the channel to express its purpose, it states that all behaviour already shown in the previous sections are intrinsic to this specialized channel.

We introduce no new entities and do not change the semantics of elements while keeping it as simple as possible. We recommend to adopt similar schemes for modelling queues and related structures as specializations of channels.

4.2.1. Push

As described before the arrow helps indicating the data flow. By comparing with the detailed version we notice that the direction of the data channels correspond to the direction of the arrow here.

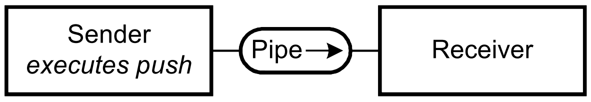


Figure 4-4 Push method, modeled simple

4.2.2. Pull

Similar to the notation for the push method we use the arrow to indicate data flow, while introducing a directed channel that is used to transmit pull requests.

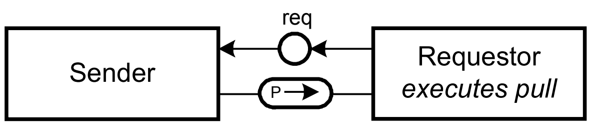


Figure 4-5 Pull method, modeled simple, "P"-shorthand-variant

4.2.3. Two active filters

The case with two active filters differs slightly. We decided not to use a similar specialized channel here or other structures as all other approaches would either be too abstract (an undirected channel) or yield contradicting semantics. Also there are good reasons to explicitly model the pipe as an actor. Its buffering and coordination capabilities are more important at these interconnections points, which often are connection points between whole filter chains, and the pipe could exhibit some special behaviour what too would justify explicit modelling. Several aspects, such as flow control or buffering, should only be modelled if necessary on an architectural level.

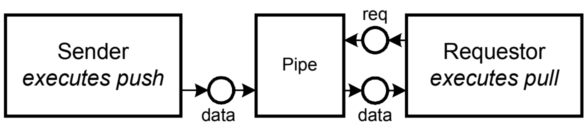


Figure 4-6 - Two active filters and a pipe

4.3. Alternative notations in FMC

We have evaluated many alternatives and dismissed them for various reasons.

4.3.1. D-sign, F-sign…

A minor modification, the change of the "P"-sign to "D" that maybe stands for "data".

We want to point out the existence of the pipe actor that is included in our detailed model. "P" should exactly express that this channel is shorthand for the detail model we presented.

Therefore no other denomination (flow, data…) was chosen.

4.3.2. Selective access

A storage between two actors being written to by using an edge labelled "add" and being read by an edge labelled "fetch" or "remove" changes the semantics of the read and write access, furthermore it is not intuitive. In addition it prohibits flow control without introducing more changed semantics or overcomplicating the model.

4.3.3. Further alternatives

We do not describe further alternatives we found because they either lacked to indicate the data flow or introduced unintuitive or overly complex models.

4.4. Existing alternative notations

We did not encounter any existing alternative approach to model the pipes and filters pattern in a reasonable way.

4.5. A complex example

Figure 4-7 depicts a more complex example of a camera with some exemplary components. The video camera chip is an active component which continuously captures data from a real world object and pushes it into the pipe next to it. The pipe buffers this image data. For simplicity the chip cannot be turned off. The command execution is another active filter similar to a processor, but unless invoked from the user who interacts with it through the user interface it has a passive role.

Once activated it becomes an active filter and pulls data through the (upper) pull pipeline. The pull pipeline consists of a configurable DSP and a zoom chip, which in turn consists of a zoom filter and a soften filter. Here you can see, that a filter can consist of several lower level filters. The filters in the pull pipeline perform their tasks depending on the request and the data pulled out of the pipe by modifying the (for example) 16kB chunks pulled from the pipe. Once a whole image (depends on selected options, for example 4096kB) has arrived at the command execution

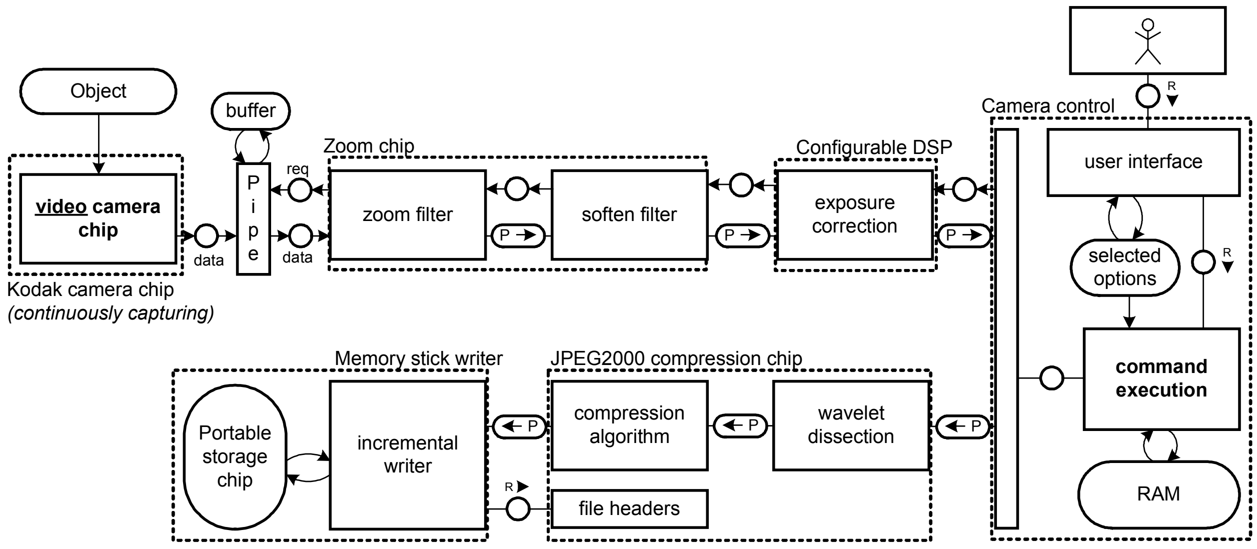


Figure 4-7 Complex Example (active components are bold-typed)

and the user issues to save it to the portable storage chip, the execution pushes the data (stored in RAM) down the push (lower) pipeline consisting of a compression chip and a smart media writer. Each filter propagates the (16kB) chunk it has been pushed to after applying some processing to it. Finally the chunks arrive at the incremental writer, which asks the compression chip what headers to write and then stores all chunks it receives on the portable storage chip (Note: If the pipe mechanism is the same between the compression chip and the writer, the chip will wait until it can fill the chunk (16kB) or all input data has been received).

This example is a modified version of the common scenario having an active component with a passive pull and a passive push chain.

Naturally extensions come into mind, for example introducing an advanced pipe (here a message router) between DSP and zoom chip to route requests to a test stub for automatic tests from which test data can be pulled. Broadcast channels or t-junctions could be used to write complete images on request to different media at once or to write logs.

As one can clearly see it is easy to determine which components have to be exchanged in order to change processing, which components can be reused in which ways, how test stubs can easily be added, how and where features (additional DSPs) can naturally be added and so on.

5. Variants and Extensions

The following variants extend Pipes and Filters by softening its principles. They allow to use Pipes and Filters in much more cases, but also they are more complicated as "standard" Pipes and Filters.

5.1. Variants

Sometimes one of the following variants can be useful.

5.1.1. Filters with more than one input/output

The linear Pipes and Filters architecture can be varied allowing filters with more than one input and/or output. Processing can then be seen as a directed graph, that can even contain feedback loops. Such systems are very flexible, but they can grow fast very complex and become hard to control. One should restrict to simple acyclic graphs, which makes complexity controllable but can be nevertheless a useful extension to filters with only one input and output.

5.1.2. Additional shared memory

Introducing additional memory, shared by all filters, is another extension. This memory can be used in three ways: as read-only source for information needed by all filters, as a global state or to improve performance.

If several or all filters need some information additional to the data they process it is suitable to use a region of shared memory. Normally, it would be more complex to pass this information along with the processing data.

A global state (see Figure 5-1) can be sometimes useful, but often, the dependencies between filters it introduces reduce the possibility to recombine filter chains. Nevertheless, in some cases it is easier to have a global state then to pass global information along with processing data, involving more complex data structures and an increase in data volume to be handled by pipes.

If filters run on one machine, shared memory can be used for a significant improvement of performance. The idea is to keep all processing data in shared memory and pass only pointers between the filters. Thus, no data must be copied, but nevertheless, it can be ensured that only one filter at a time accesses a chunk of data.

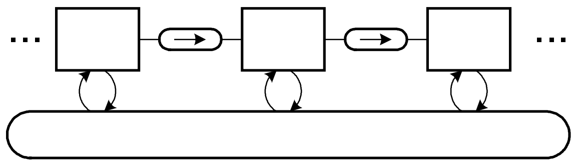


Figure 5-1 Several Filters sharing a global state

5.1.3. Advanced Pipes

A Pipe normally is a relatively simple component, consisting mainly of a buffer and methods to put data into it and to remove data from it. A standard example are UNIX pipes. The most simple implementation of a pipe, when connecting an active with a passive filter or two passive filters, can be a simple procedure call. But also very sophisticated pipes can be imagined. In fact, every mechanism that can be used to transport data from one system component to another can be used as a pipe. For example, we can think of broadcasting pipes or routing pipes, that do not only buffer and transmit data, but decide based on environment data or the processing data itself where to send it. Another possibility could be a pipe with a publisher/subscriber mechanism, sending different data depending on its type to all components subscribed for this type. Such complex pipes allow a huge variety of applications, but we loose the benefits a uniform pipe interface introduced, for example easy recombination.

5.2. Dynamic filter chains

Filter chains can either be static structures, depending only on their source code, or they can be created dynamically (see Figure 5-2). Using the configurator pattern can be appropriate here. The structure of a filter chain can depend on some configuration data (e.g. a config file) or on the data processed (see section 6 for an example).

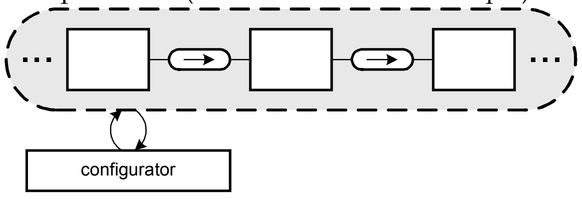


Figure 5-2 A dynamic filter chain

6. Exemplary implementation: Apache 2 Web Server

Apache 2 intensively uses filters to process requests. For each request, a pulling pipeline (called input filter chain) and a pushing pipeline (called output filter chain) is created. Both filter chains consist only of passive filters. Once these filter chains are created, the request processing component pulls request data from the input filter chain and pushes response data to the output filter chain. Each filter and the request processing component can modify the data.

Apache divides each request or response into small chunks, called "buckets", and some buckets are held together in one brigade (see Figure 6-2). The filters work on a brigade at a time, pass it to the next filter and then work on the next brigade (if available)

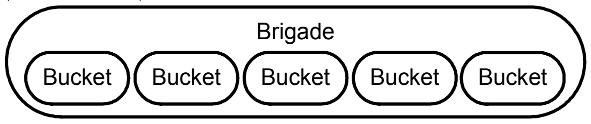


Figure 6-2 A Brigade containing several buckets

The brigades lie in a shared memory, so filters do not pass brigades, but actually pointers to brigades to improve performance (see Figure 6-3).

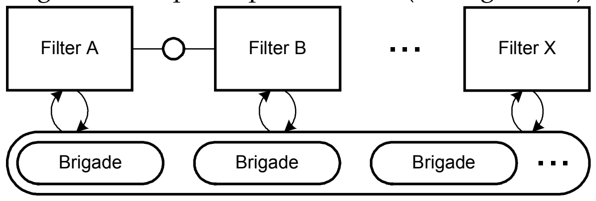


Figure 6-3 Filters access brigades lying in a shared memory area

7. Appendix – Distinction from other architecture patterns

Pipes and Filters can be separated relatively clear from the following patterns, because it is the only pattern that addresses processing of large amounts of data and focuses ease of recombination and reuse of processing steps.

- Broker
  The Broker pattern is a way of decoupling point to point communication.
- Leader/Follower, Half-Sync/Half-Async, Reactor
  These patterns deal with the problem of handling concurrent events
- Interceptor

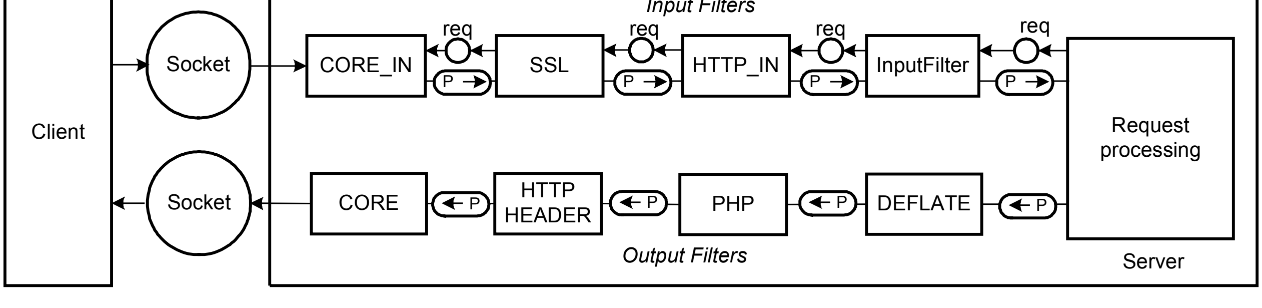


Figure 6-1 Apache Web Server Request Processing

Interceptor introduces a possibility to delegate processing of unprocessable events to "plugins" dynamically

- Microkernel
  Microkernel describes a specific structure of systems.

References

[AMP] The Apache Modeling Project, http://apache.hpi.uni-potsdam.de

[POSA2000] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-Oriented Software Architecture. Wiley, 2000

[EIP] Enterprise Integration Patterns, http://www.enterpriseintegrationpatterns.com

[AP2SRC] Apache 2 Web Server Source Code, http://httpd.apache.org

[AP2DOC] Apache 2 Documentation, http://httpd.apache.org/docs-2.0

[AP2M2002] A. Langhorst, M. Steinle, Apache 2 Modules, Lecture in Seminar Systemmodellierung, 2002

Modeling of the Broker Architectural Framework

Konrad Hübner, Einar Lück
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60
14440 Potsdam, Germany
{Konrad.Huebner,Einar.Lueck}@student.hpi.uni-potsdam.de

Abstract

The Broker Architectural Framework describes how distributed systems can be structured in order to achieve location, platform and language transparency. The use of this pattern allows architects to structure their applications in a way that makes it much easier to reuse existing solutions and organize system architectures in a way that increases the overall maintainability.

This paper presents an approach towards modeling this pattern that goes beyond the original description in [1] aiming at an improved knowledge transfer at the architectural level. Instead of Class-Responsibility-Collaborators-Modeling and class diagrams, Fundamental Modeling Concepts (FMC) are used in order to describe the structure and the key benefits of this pattern.

1. Introduction

In nearly every engineering discipline the reuse of existing solutions for difficult problems that have proved their value is essential to speed up product development cycles.

Software engineers have to cope with the problem that arises from the fact that solutions for different products are developed with different programming languages and due to the lack of interoperability cannot be reused easily. Lack of interoperability is also a major concern if different operating systems or platforms are utilized or different protocols serve as basis for communication in distributed systems. We conclude that in order to circumvent these problems it is necessary to achieve language and platform transparency from the user's point of view.

An engineer does not only have the task to construct solutions in a way they can be reused easily, he is also responsible for designing systems as maintainable as possible in order to avoid high maintainance costs. In distributed systems the overhead of code changes necessary in case of changes in the service locations decreases the maintainability of systems. Obviously, a lack of location transparency is the reason for this problem. A related major concern implied by maintainbility is that existing systems should be reconfigurable at runtime. Therefore the systems have to be partitioned in parts that can easily be replaced. Many systems lack a standardized infrastructure that allows the solution of the stated problem.

The Broker Pattern [1] addresses the mentioned reusability and maintainability requirements by proposing an architectural pattern in which these concerns are encapsulated in a way that full language, platform and location transparency is achieved. In addition to that the decomposition of functionality in reusable, exchangeable and reconfigurable entities is enforced.

The following section describes the solution structure of the pattern using the Fundamental Modeling Concepts [2] and goes beyond the original description in [1]. The chosen notation aims at communicating the idea and the concepts of the pattern as efficiently and comprehensible as possible.

The discussion section is dedicated to review our description approach and compares it with the original description in [1].

2 The Broker Pattern

2.1 Proposed Description

To fulfill the requirements listed in the previous section, several components have to be introduced. Put together, they form the Broker Pattern. The first introduced component to be added to the system is the Broker. It realizes the distribution of requests to the responsible components and the transfer of results and exceptions and thus supports the detachment of functional components from the system to break up complex structures. The Broker is placed between components and clients as depicted in Figure 1. As a result, functional behaviour can be added, replaced or removed easily because clients are not affected. By placing the Broker between client and components, location transparency can be implemented.

During the implementation of location transparancy it comes obvious that it is necessary to associates names with components. That's the reason why some kind of naming service has to be implemented that becomes part of the broker. It is important to understand that the associated names contain no information about the location of the component like IP-Addresses or network names or something like that.

Ideally, all communication traffic has to pass the Broker as shown in figure 1. This might result in a bottleneck at the Broker. To circumvent this problem it is reasonable to soften the concept of completely encapsulated communication to some extent depending on the context.

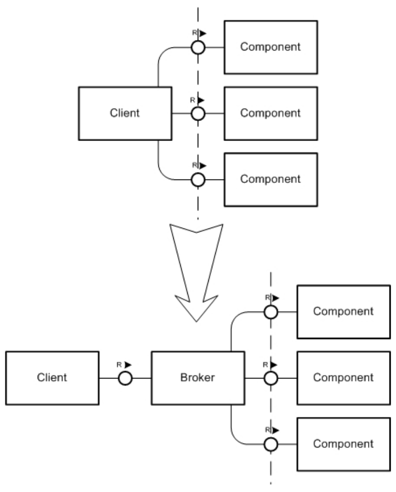


Figure 1. Broker component

The idea is that we have one Broker connecting all components and clients. It is obvious, that each client or component needs a local communication gateway, so the abstract Broker is separated into local Broker for each connected system.

The next requirement, encapsulation of Broker access, is implemented through proxies. As a result, the client does not need to know anything about inter-broker-communication and Broker implementation details. Same issues apply for server components which access the Broker through a proxy as well (Figure 2).

The use of proxies supports language transparency. For example, proxies can be generated automatically from language-independent interface descriptions (IDL). Each supported language has to provide a compiler for IDL documents in order to generate proxies and stubs. The requirement for this strategy is that a mapping between the implementation language of the Broker and the target language exists.

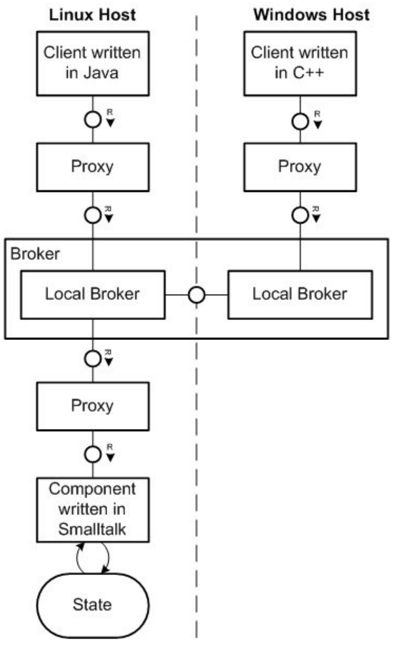


Figure 2. Broker with proxies

Another way is to define a binary layout of method tables. The binary method table standard has to be supported by the target programming language.

The aspect of inter-broker-communication needs to be addressed at this point. With a homogenious Broker network, we should not need to worry about inter-broker-communication, because the implementation will cover this point. The Broker Pattern also conciders heterogenious systems due to network- or broker-specific incompatibilities. The solution to circumvent this problem is the introduction of a bridge component, encapsulating system specific network details and resolving Broker incompatibility by message conversion. The bridge component is necessary for complete platform transparency.

With these components - Broker, Proxy and Bridge - all needed parts to realize a Broker Pattern for platform-, language- and location-transparent communication exist. Figure 3 shows a compositional structure diagram combining all introduced parts.

The goal of better maintainability of a distributed system is reached by the encapsulation through the Broker. Components can be registered and deregistered at the Broker (add and remove). The replacement of components is not specified by the pattern. Special concerns are necessary because requests during the exchange phase have to be delayed until the new component is available. Moreover, the current state of a component might be important and therefore has to be saved before the component is replaced to keep the system

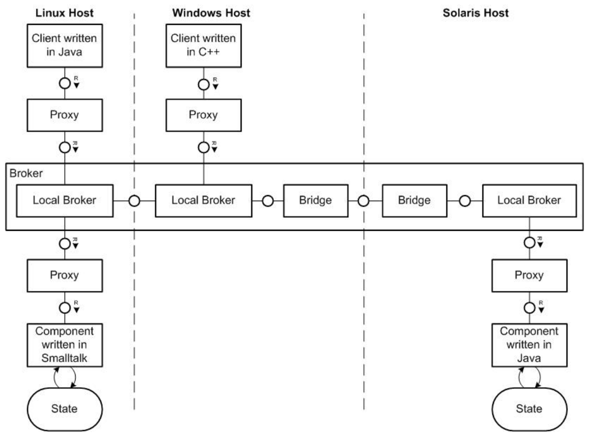


Figure 3. Broker with proxies and bridges

consistent.

As mentioned above, the pattern can be softened concerning the communication ways. There are two different variants of the pattern: The indirect and the direct communication approach.

The indirect communication approach requires all requests to pass the Broker, so that a complete encapsulation of the entire communication process is reached. None of the components and clients has to know where and on what system the other ones reside.

To increase system performance the Broker component can be disburdoned by establishing a direct connection between two components or a client and a component, after the Broker has found the component responsible for the current request. A resulting requirement is that the client and the component which serves the request use the same protocol, otherwise they could not communicate with each other.

The dynamics of a typical request from the client via the Broker to the component and back in a homogenious Broker environment (no bridges needed) are shown in figure 4.

2.2 Applications of the Broker Pattern

The typical application field for the Broker Pattern is a middleware environment. The purpose of middleware systems is to hide aspects of distribution from the appication. The CORBA implementation is closest to the Broker Pattern of all middleware platforms. This is no surprise as the pattern was described after CORBA was implemented. Other middleware platforms contain elements of the Broker Pattern and a mapping to the main components - Broker, Local Broker and Proxies - can be made. We also inspected language transparency and communication means.

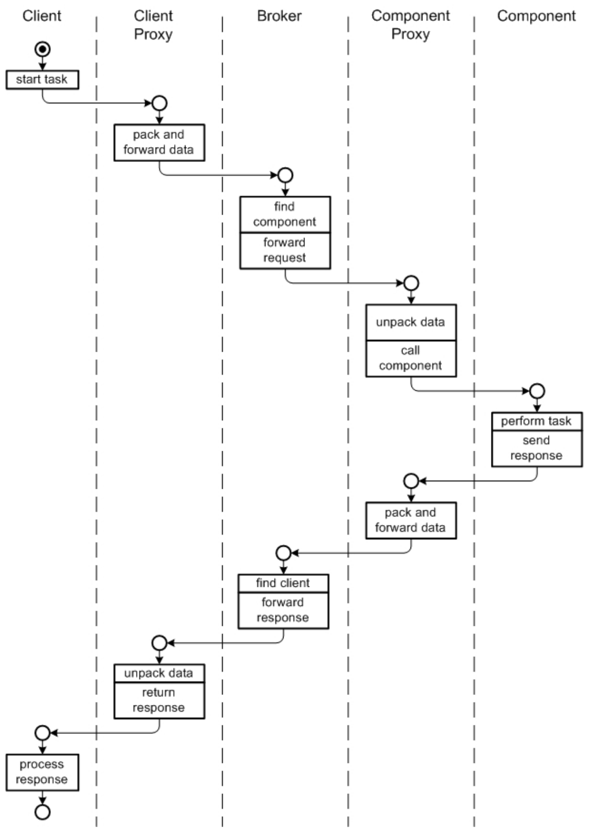


Figure 4. Broker dynamics

2.2.1 CORBA

The best-fitting example for an application of the Broker Pattern is CORBA [4]. In CORBA, a so called Object-Request-Broker (ORB) is running on each connected system. The ORB ist responsible for the discovery of components and distribution of requests. In this it implements the Local Broker on each system. According to our pattern description the sum of all ORB is the conceptual Broker.

CORBA programmer's have to describe the interfaces of their components in the Interface Definition Language (IDL) that is independent from implementation languages like Java, C++ and C. Client and server proxies are generated by the correpsonding IDL compiler and are called IDL stubs and skeletons in terms of CORBA. Obviously IDL is an important tool to achieve the stated language transparency. All we need in order to interact with a certain service is the a mapping from the interface description in IDL to the programming language with which we describe our client.

As we pointed out in the pattern description it is necessary to map names to components. For each component registered with an ORB a so called Interoperable-Object-

Referenece (IOR) is created in which the address of the hos of the component, a unique number and further information that is necessary to discover the component is encoded. This IOR is sufficient for discovering an object. But IORs are not human readable, encode information about the location of the component and in this do not fulfill our requirements on naming. The CORBA Naming Service is a component that has to be located on one host in the network. During startup of the Local ORB a reference to the CORBA Naming Service is passed to it. The Naming Service is responsible for mapping human readable names to IORs and provides an appropriate CORBA interface.

Platform transparency is also addressed seriously by CORBA. For the communication between ORBs, IIOP and GIOP (Internet / General InterOrb Protocol) are specified precisely.

In figure 5 you can see the structure of an example CORBA system. The naming service aspects are left out in order to ease the mapping to our pattern description.

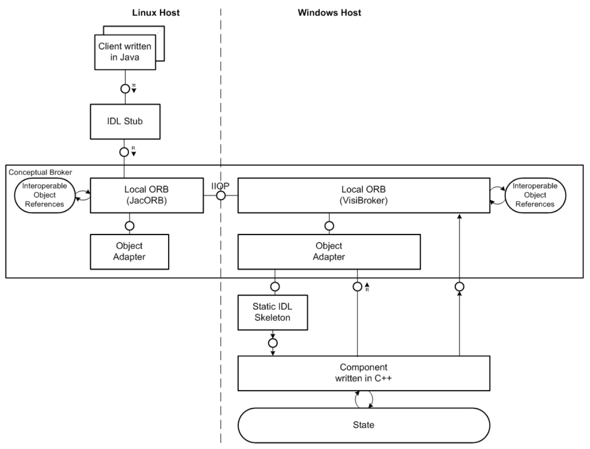


Figure 5. CORBA system structure

2.2.2 DCOM

The DCOM environment [5] from Microsoft consists of much more components building the middleware infrastructure than CORBA. That's the reason why the mapping to the Broker Pattern is far more complex.

DCOM specifies that clients primarily interact with the so calle COM Service Library in order to connect to a certain component. The latter communicates with the Service Control Manager (SCM) in order to locate the component the client wants to interact with. The SCM checks wether the component is a locally available. This is implemented by contacting the OXID-Resolver which contacts the OXID-Resolvers located on other hosts in order to discover the desired component if necessary. After this the COM Service Library has the function to instantiate a Proxy either a one for local Inter-Process-Communication or for communication via network. In both cases the Proxy interacts via the Prox Manager with the component. In case of distributed components the corresponding channel is creates by the OXID-Resolver. Aware of our stated pattern description we can now conclude that these 4 agents form the Local Broker.

For each type of component a so called Class-Identifier (CLSID) and a Class Factory are associated. The Class Factory is responsible for creating an instance of the component. Thus the issue of naming is addressed by the formerly stated Service Control Manager (SCM) who maps CLSIDs to Class Factories.

In order to achieve language transparency Microsoft has specified binary method table standard. As stated in the pattern description this approach requires all programming languages to support it. This is for example the case for C++, C and Visual Basic.

For interoperability between different host types the specification of communication protocols is based on an extended version of the Distributed Computing Environment's (DCE) Remote Procedure Call (RPC). Thereby platform transparency is implemented.

A simplified compositional structure of the DCOM environment is depicted in Figure 6.

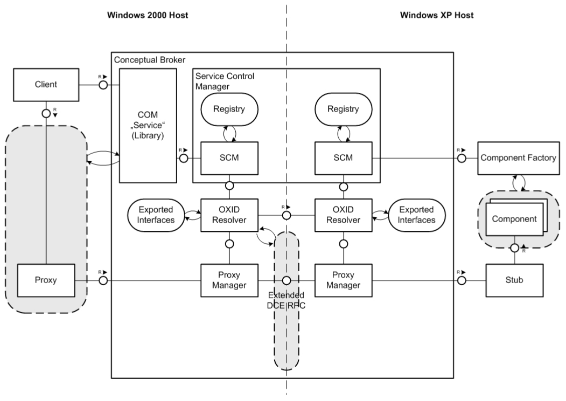


Figure 6. DCOM system structure

2.2.3 Enterprise Java Beans

The third inspected middleware platform, Enterprise Java Beans (EJB) [6], is very complex and the connection to the Broker Pattern is by far not as obvious as in the case of CORBA. Nevertheless, it is possible to identify participating components and map them to the Broker Pattern. Naming services are provided by the JNDI service. It enables an application to look up services supplied by an EJB Server and connect to them. The EJB server can be considered as the Broker. Proxies that implement home and remote interfaces reside within the server. On the client side the corresponding element is the stub. EJB is bound to Java, so there is no native language transparency. This aim is reached by CORBA support. As a result all CORBA services can be integrated into an EJB environment. JavaRMI is used as the communication protocol between client and server. Figure 7 shows a very abstract view of the EJB architecture. EJB is far more complex and an in depth description of it goes far beyond the scope of this paper.

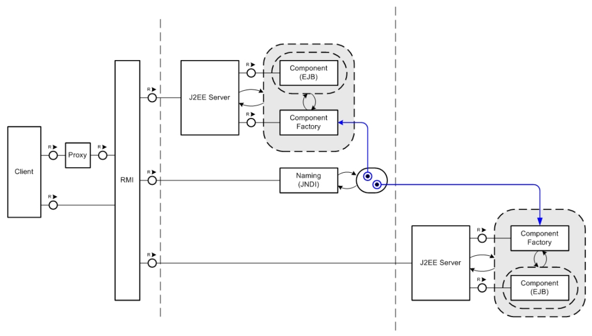


Figure 7. EJB system structure

2.3 Relation to other patterns

In this section we describe in which way the pattern is related to other patterns from [1] and [3].

2.3.1 MicroKernel

The MicroKernel Pattern [1] aims at organizing the architecture of a system in a way that just the minimal core functionality of the system under construction is put into an architectural entity and all other architectural entities use this core in order to fulfill extended services. The relation to the Broker Pattern consists of the idea of the modularisation and exchangability of components but the MicroKernel Pattern does not take into account the aspects of transparency and configurability like the Broker Pattern does.

2.3.2 Component Configurator

This pattern, taken from [3], is in one aspect related to the Broker Pattern. It describes mechanisms that allow the configuration, addition, replacement and removals of components at runtime. The former subsections state in which way the Broker Pattern addresses this issue. Location, platform and language transparency issues are not addressed explicitly.

2.3.3 Interceptor

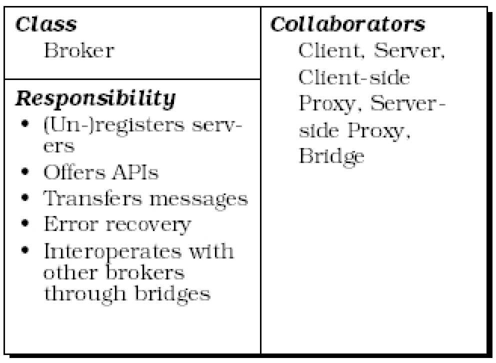
The Interceptor Pattern [1] addresses the issue of transparent enhacement of existing systems with additional functionality. It thus takes into account the issues of extendability and transparency. The Interceptor Pattern focuses on an event-driven approach towards the activation of services. In contrast to this, the Broker Pattern focuses more on location, platform and language transparency, aspects not covered by the Interceptor Pattern.

2.3.4 Forwarder-Receiver

The Forwarder-Receiver Pattern is strongly related to the Broker Pattern. It addresses the issue of Peer-To-Peer communication. The focus is set to the abstraction from the inter-process communication protocol. It lacks solutions for the platform, language and location transparency problem. Proxies that hide the distribution aspect are for example not part of the pattern description.

3. Discussion of the description

The original description in [1] divides the solution structure of the pattern into classes. For each class responsibilities are collected and stated precisely. In addition to that other classes a certain class has to corporate with in order to fulfill its service are identified. These aspects are written down on so called Class-Responsibility-Collaborators-Cards (CRC-Cards) - Figure 8. We consider this to be very useful in order to understand the pattern in detail. Relationships and the operations of each of the classes are expressed by an Object-Modeling-Technique-diagram (OMT-diagram) - Figure 9. Together with the detailed verbal description one can grasp the nature of the pattern and is enabled to apply it.



Class Broker	***Collaborators*** Client, Server, Client-side Proxy, Server-side Proxy, Bridge
Responsibility • (Un-)registers servers • Offers APIs • Transfers messages • Error recovery • Interoperates with other brokers through bridges	

Figure 8. CRC-Card for the Broker class taken from the orginal Broker Pattern description

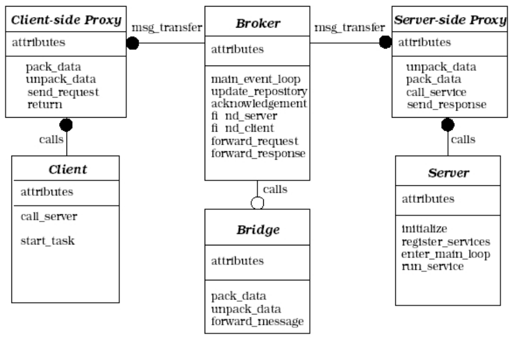


Figure 9. OMT-diagram taken from the original Broker Pattern description

We think our approach complements the original description. The application of the pattern has strong implications on the compositional structure of a system. This aspect is not addressed in the original description. Our intention is to focus the description on this issue and thereby improve the overall comprehensibility of the pattern description. We consider this to be useful because we think especially in order to understand the issues of distribution, communication and (location, platform and language) transparency it is appropriate to consider this dimension. The block diagram notation introduced in [Keller et al.] allows the visualization of communication relationships between agents in the compositional structure of a system in a certain point of time and enables us to depict these aspects in Figure 2.

Interpreting Figure 2 we think it is easy to grasp that each Client uses proxies to interact with services located on other machines written in different languages transparently. Aware of the overall context of the pattern the responsibilities of the Broker and the function and the responsibilites of its refinement into Local Brokers can be concluded, too. We do not want to pledge for a reduction of the textual description of patterns. Instead we aim at an improved comprehensibility by introducing additional pictures.

Considering Figure 5, Figure 6 and Figure 7 we conclude that a concrete application of the pattern and its compositional structure can be depicted in an easy to comprehend way. In addition to that it is also possible to recognize the pattern application in concrete systems in case compositional structures are visualized through the use of block diagrams.

We consider it to be problematic to structure the pattern in terms of classes. Classes are normally associated with the Object-Oriented development paradigm. As we know from [CORBA] it is also possible to apply this pattern in case of procedural languages like C. Aware of this fact we consider an illustration in terms of agents with certain responsibilities to be more appropriate.

4. Conclusion

The Broker Architectural Framework has strong implications on the compositional structure of systems that are not addressed in the description of the pattern in [1]. That's the reason why we have proposed an enhanced description based on Fundamental Modeling Concepts [2] that emphasizes this dimension of the pattern.

Furthermore this modeling approach can be utilized to visualize applications of patterns as we have shown for EJB, DCOM and CORBA. We pointed out that in order to understand the certain aspects of the pattern like location, platform and language transparency we consider it to be essential to focus on the compositional structure of systems.

Our description approach is a step towards more comprehensible pattern descriptions and complements existing pattern descriptions.

References

[1] F. Buschman, R. Meunier, H. Rohnert, *A System of Patterns. Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996

[2] F. Keller, P. Tabeling, R. Apfelbacher, B. Gröne, A. Knöpfel, R. Kugel, O. Schmidt, *Improving Knowledge Transfer at the Architectural Level: Concepts and Notations*, Proceedings of The 2002 International Conference on Software Engineering Research and Practice, Las Vegas, 2002

[3] E. Gamma, R. Helm, R. Johnson *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, 1997

[4] http://www.corba.org/

[5] http://www.microsoft.com/com/tech/dcom.asp

[6] http://java.sun.com/j2ee/

Microkernel – An Architecture Pattern

Eiko Büttner, Stefan Richter
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{eiko.buettner, stefan.richter}@student.hpi.uni-potsdam.de

Abstract

The microkernel architecture pattern can be applied to software systems that must be able to cope with changing requirements. This pattern promotes an architecture, where a minimal functional core serves as an abstraction layer for underlying hard- or software and as a socket for plugging in components that offer extended or customer-specific functionality. Thus, this pattern fosters easy portability as well as changeability and extensibility at the cost of a high degree of engineering complexity.

This article describes the microkernel architecture pattern based on [POSA1996]. It shows how this pattern can be used for developing software systems that need to run on multiple hard- or software platforms and that can be extended in an easy manner.

1. Introduction

Originally, the concept of kernels was developed in connection with the modularization of operating systems. The term kernel refers to the relatively small but extensible core functionality that all well-structured operating systems provide in contrast to monolithic ones. An operating system is said to have a microkernel if its architecture follows "an approach to operating system design emphasizing small modules that implement the basic features of the system kernel and can be flexibly configured" [FOLDOC].

However, this definition reveals an important problem[1] when it is used for classifying systems: how small are these modules exactly and how are basic features separated from all other features that have to be implemented as well? Hence, for a clear classification it is very important from what point of view a system is analyzed.

Furthermore, this definition focuses on operating systems only, although the microkernel architecture pattern can be applied to other software systems as well. This follows the common software development paradigm of designing small code units implementing a clear-cut functionality. For example, a microkernel middleware platform (OSA+[2]) has been developed at the University of Karlsruhe, and Pervasive Software is selling a database management system based on a microkernel architecture – the MicroKernel Database Engine (MKDE).

The following aspects have to be taken into account for a characterization of microkernel systems:

- Memory footprint: An easy definition, that focuses on this rather trivial aspect could be as follows: "A kernel is a microkernel if its memory footprint is smaller than x KB." This aspect – however simple it may be – is considered surprisingly often. Especially when it comes to embedded systems, the size of a microkernel is of utmost importance, given the limited resources these systems generally provide.
  Obviously, this definition has to be changed depending on a system's functionality and field of application. For instance, the memory footprint of the monolithic operating system MSDOS 1.0 is certainly smaller than the one of the real-time microkernel operating system QNX 6.0.
- Reliability and security: This aspect is based on the observation that a system is more reliable (i.e. more tolerant towards programs or modules containing errors and more resistant to system attacks) if its (micro)kernel contains only basic functionality that is used by all other system components as well as functionality

[1] Many other definitions have the same or similar problems.

[2] OSA+: *O*pen *S*ystem *A*rchitecture - *Pl*atform for *U*niversal *S*ervices

that needs special access rights and thus cannot reasonably be implemented elsewhere. Considering that functionality with special access rights generally poses a threat to system stability (since it could potentially damage any system component), it is an especially good idea to implement it in the microkernel as this system component is usually the one the most extensively tested. Furthermore, faulty system components with limited access rights are less likely to crash the whole system.

- Portability and decoupling of system components: Within the scope of reusing source code or even compiled binaries this aspect is becoming more and more important. It aims at making system components as independent as possible from underlying hard- or software and from each other.
  In microkernel systems, many components are typically implemented as separate processes that build on the basic functionality offered by the microkernel. The microkernel on the other hand exports abstract and generic interfaces to most parts of the underlying hard- or software platform, thus serving as an abstraction layer. Ideally, only a system's microkernel has to be adapted when it is ported to a new platform.
- Scalability and configurability: In many microkernel systems the microkernel serves as a socket for plugging in components offering extended functionality such as device drivers or support for network communication protocols. The microkernel system can be flexibly configured by adding only required components and removing all others.

With all these different aspects in mind, it should be a rather difficult task to find one concise definition describing all possible types of microkernel systems. Most system architectures will concentrate only on one or a combination of some of the aforementioned aspects, paying less attention to others.

A better understanding of the microkernel concept may be achieved by putting all aspects into a pattern – the microkernel architecture pattern. The pattern described in this article helps developing new software systems based on a microkernel architecture as well as understanding already existing ones.

2. The Microkernel Architecture Pattern

2.1. Participating components

According to [POSA1996], the microkernel architecture pattern defines five kinds of participating components: the microkernel, system services, system views, adapters and clients. Note that in certain microkernel systems some of these components (mainly adapters and system views) might be absent.

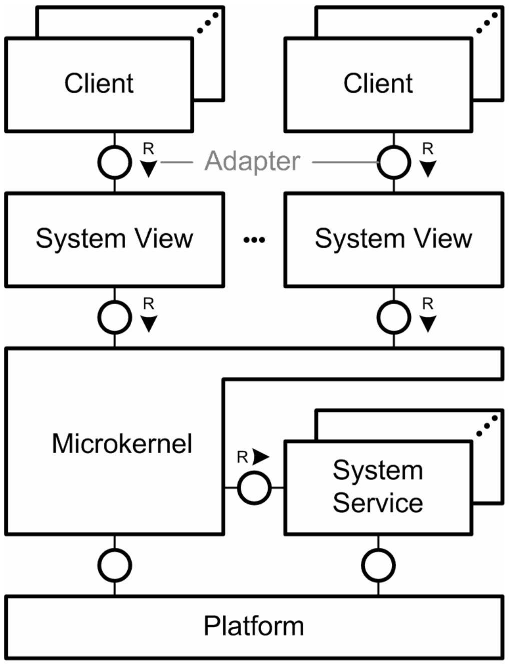


Figure 1 Structure of a microkernel system.

2.1.1. The Microkernel

The *microkernel* is – as expected – the central part of a microkernel system. As pointed out before, it encapsulates basic functionality needed by all system components as well as such functionality that absolutely needs the special access rights that a (micro)kernel typically has and that would affect system stability, security or other important aspects. It offers abstract interfaces to the underlying hard- or software platform, providing atomic services that can be used by other system components to implement more complex services.

When designing a microkernel, special care should be taken that the amount of functions implemented in it remains as small as possible. Mainly, because otherwise benefits such as easy portability, maintainability and changeability will be lost, and in fact a microkernel with an overwhelming amount of functionality does not deserve its name anymore.

A typical microkernel is responsible for managing resources and enabling other system components to communicate with each other. In [TanVsTor1992] Tanenbaum points out, that an operating system microkernel should handle inter-process communication (IPC), interrupt requests (IRQ), low-level process management and possibly IO[1]. Operating system microkernels normally use the IPC services they provide for exporting their programming interfaces.

2.1.2. System Services

System services[2] extend the microkernel's functionality. The microkernel serves as a socket for plugging in such system services either statically at compile-time or dynamically during startup or even at run-time. This helps keeping the memory footprint of a microkernel system as small as possible and making it scalable and configurable.

As shown in figure 1, system services are accessible through the microkernel only. When the microkernel receives a request from another system component, it decides whether it can handle this request directly or whether it needs a system service for this purpose. In the latter case, the request is transparently forwarded to an appropriate system service. Thus, other system components are generally unaware of where and how certain functionality is implemented.

As a system service provides extended functionality that is not directly offered by the microkernel it might also be dependent on the underlying hard- or software platform. In operating systems, system services are commonly used for implementing device drivers and page fault handlers. OSA+ uses system services to offer additional functionality such as event logging and security services and to provide interfaces to an underlying operating system [OSA+2000] (see also figure 6).

2.1.3. System Views

System views[3] implement different views of the system and introduce a further abstraction layer into the system architecture by using the microkernel's atomic services to form more complex services. Each system view is normally implemented as a separate (user-mode) process.

Especially in operating systems, system views export their interfaces in the same way the microkernel does (e.g. by using IPC). In other systems, system views might either rely on the microkernel or on functionality provided by an underlying operating system (e.g. network communication or IPC).

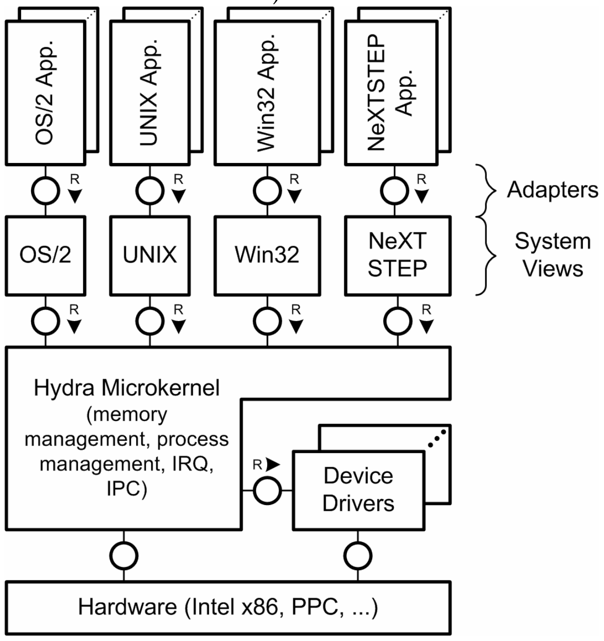


Figure 2 Structure of Hydra – a conceptual microkernel OS: Note that in [POSA1996] it is pointed out that Hydra's adapters need to access the microkernel in order to establish IPC connections to their associated system views. This is not shown in this diagram as the primary purpose of adapters is to provide a communication channel between applications and system views.

Operating systems typically use system views to emulate various full-fledged operating systems built on top of the microkernel. In [POSA1996] a conceptual operating system called Hydra is taken as an example of how to implement a microkernel architecture system. Hydra uses system views to provide the programming interfaces and functionality of a set of already existing operating systems such as UNIX System V, OS/2, MS Windows and NeXTSETP. Each of these system views runs in a separate process, exposing its API by means of IPC facilities provided by the microkernel. The Pervasive MKDE uses system views to provide different data models of its physical data. Currently, it features transactional (Btrieve) and relational (Pervasice.SQL) models [PERVPROD] (see also

[1] In other articles Tanenbaum mentions basic memory management as well.
[2] In [POSA1996] these are called "internal servers".
[3] In [POSA1996} these are called "external servers".

figure 7).

2.1.4. Clients

Clients are applications (or application-like modules) that employ the functionality one or more system views expose through their interfaces. In case of operating systems, a client is usually associated with exactly one system view. E.g. in Hydra, MS Windows clients would be restricted to using the MS Windows system view while an OS/2 client would use the OS/2 system view. Hydra does not allow one client to be associated with more than one system view. Other microkernel systems however might allow clients to access multiple system views at the same time if necessary.

2.1.5. Adapters

In some cases, an undesirably strong coupling of components may result from an architecture where clients access system views directly. For example, an application that accesses a database directly would be restricted to some specific database implementation. Using an ODBC[1] driver that offers uniform access to relational databases would enable the application to cooperate with any database management system supporting ODBC (such as the MKDE…). In the context of the microkernel architecture pattern, such decoupling components are referred to as *adapters*.

In Hydra, adapters make applications unaware of whether they are really running on the operating system they were originally developed for, or whether they are running on Hydra, using one of the system views. If applications were to access Hydra's system views directly they would have to be modified in order to employ IPC mechanisms instead of conventional procedure calls for gaining access to operating system functions. With adapters, clients only have to be recompiled. Furthermore, adapters make it possible that both clients and system views can be changed independently.

Adapters are normally implemented as libraries that are linked to clients and thus reside in their address spaces.

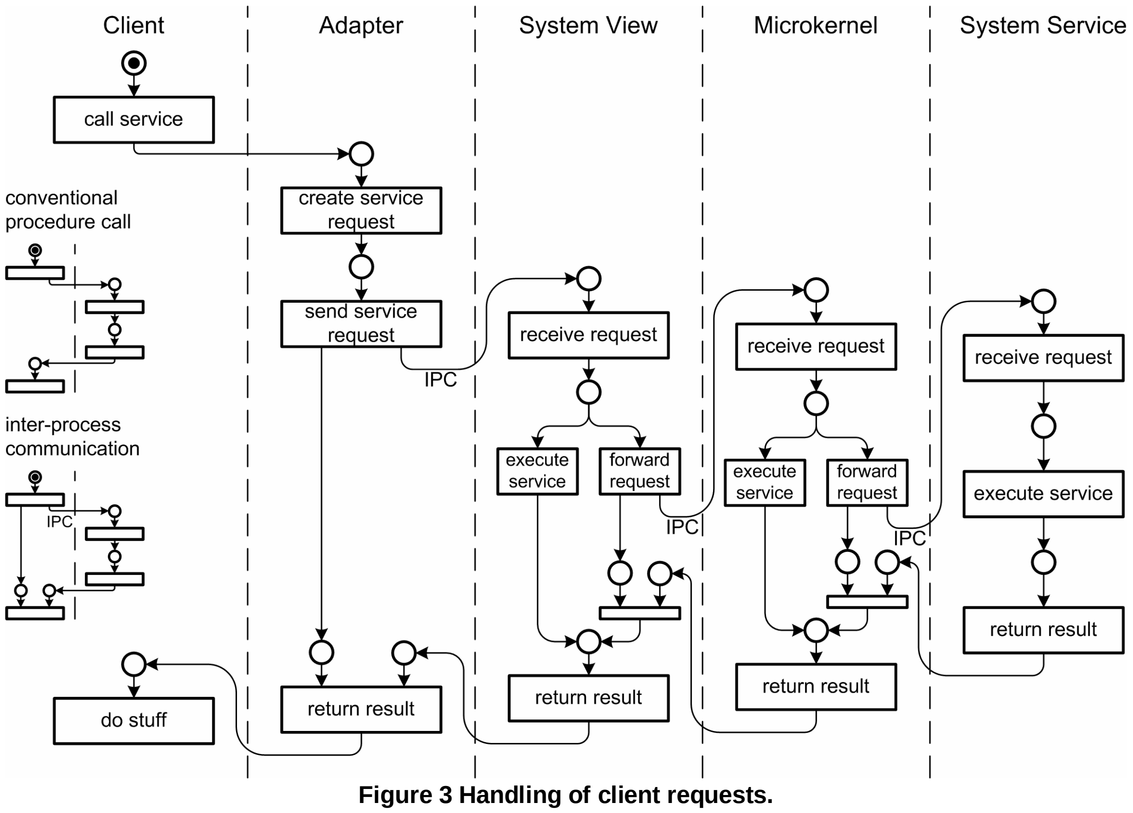


Figure 3 Handling of client requests.

[1] ODBC: Open Database Connectivity

2.2. Cooperation of microkernel components

The manner in which the various components of a microkernel system communicate exactly depends on how these components are implemented and what kind of communication facilities are available. Figure 3 shows how a client request might be handled in a pure microkernel system. While clients, system views, the microkernel and system services are all implemented as separate processes, adapters are linked to clients. Thus, clients access adapters via conventional procedure calls.

When a client requests a service of the underlying microkernel system, it invokes an appropriate routine of its adapter. The adapter then creates and sends a service request message to a system view.

That system view might either be able to handle the request directly or it might need functionality provided by the microkernel for this purpose. In the latter case the request is forwarded to the microkernel. (The system view might also send several requests to the microkernel to service the client request.)

Subsequently, the microkernel has to decide whether it is able to handle the request or whether it must invoke a system service.

As shown in figure 3, a microkernel system typically makes heavy use of inter-process communication. Especially in an operating system, where it would be the microkernel's task to handle inter-process communication, the microkernel would be activated each time an IPC message is sent from one process to another one.

2.3. Implementation

Generally, designing and implementing a microkernel system is a very complex process. Often, many different solutions may exist for a given task. Common problems are:

- It has to be decided whether all components must really be implemented as separate processes or whether some of them can be combined with other components, e.g. in order to improve overall performance. For instance, certain system services might be linked to the microkernel, or system views could be linked to clients. In small systems, even the microkernel could be directly linked to clients (The VRTX real-time operating system is an example of a microkernel that is directly linked to clients (i.e. applications) [VRTX]). Using processes normally results in more overhead because IPC mechanisms have to be employed instead of conventional procedure calls. On the other hand, combining components may decrease system stability as a faulty component can potentially damage many others, causing the whole system to crash.
- Efficient request handling strategies have to be found for those components implemented as separate processes. In particular, the most frequently used components could improve their performance by using multiple threads for servicing client requests.
- It has to be determined how much and which functionality needs to go into the microkernel, and what can be implemented elsewhere. A small microkernel might not offer enough functionality for some purposes, thus complicating the development of clients. Implementing more functionality in the microkernel may decrease portability and maintainability on the other hand.

For analyzing and designing a microkernel system, [POSA1996] proposes a top-down approach.

In a first step, a requirements analysis is performed by examining the application domain (i.e. potential clients and system views). This helps to find out what kind of functionality the microkernel system will have to offer. The results of this analysis are then grouped into semantically independent categories. In a next step, it is decided which of these functionality categories are implemented in the microkernel and what is put into system services. This decision depends on the application domain, of course.

Once it has been determined what kind of functionality the microkernel system is going to provide, and once this functionality has been distributed among the microkernel and its system services in a consistent way, the microkernel and the system services themselves have to be designed. E.g. the *layers pattern* can be applied for designing all system components serving as an abstraction layer for underlying hard- or software.

The implementation is done using a bottom-up approach. First, the microkernel and the system services are implemented. This should normally be done in parallel, as some parts of the microkernel might rely on system services, and most system services will certainly need the microkernel to work. Afterwards, everything else is implemented in the following order: system views, adapters, clients.

3. Known Uses

Microkernels have been successfully used for implementing a broad range of operating systems. Examples are Mach, QNX, Amoeba, LynxOS, VRTX, Chorus and many others. Moreover, the middleware platform OSA+ is based on a microkernel as well as the MicroKernel Database Engine.

3.1. The QNX Real-Time Operating System

The real-time operating system QNX is a good example of a strict microkernel architecture, satisfying all aspects mentioned at the beginning of this article: its memory footprint is very small, it is very reliable and secure, it is easily portable to different hardware platforms, and it can be scaled down to run on hardware with limited resources such as microcontrollers.

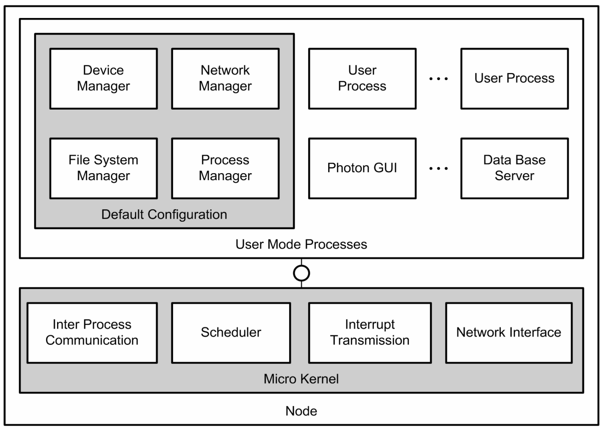


Figure 4 Structure of QNX.

This is achieved by consequently minimizing the functionality implemented in the kernel. The kernel only comprises such functions as scheduling, inter-process communication and IRQ transmission, all of which normally need kernel-mode access rights. Moreover, the QNX kernel has a network communication interface to provide support for clustering (i.e. connecting multiple computers over a network and using them as if they were one giant computer[1].)

The process manager is the only process that also has certain special access rights. All other processes – device drivers, services and applications – run in user-mode. This is an important aspect as technically there is no difference between system services (device drivers…) and applications. They just fulfill different functions.

QNX (this abbreviation stands for Quick UNIX) has only one system view – a UNIX-like API. This system view is linked to clients at compile time like an adapter.

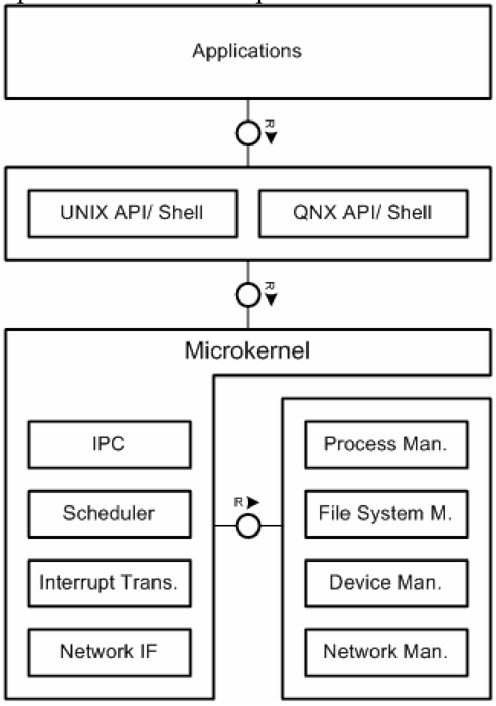


Figure 5 Structure of QNX mapped on pattern

The main inter-process communication mechanism in QNX is the so called message passing. If one process wants to send a message to another process the kernel is invoked in order to copy that message from the address space of the sender into the address space of the receiver.

Figure 6 shows how synchronous inter-process communication is implemented in QNX. After having invoked the Send() routine the sending process remains in blocked states (SEND and REPLY) until it receives a response.

The receiving process on the other hand invokes the Receive() routine in order to signal that it is prepared for receiving messages. It remains blocked (RECEIVE) until a message arrives. As soon as it receives a message it may work on so as to handle the request.

For sending its response, the receiving process invokes the Reply() routine. At this point, there is

[1] In [POSA1996] this is called *distributed microkernel*.

no need to put this process into a blocked state since kernel activity automatically blocks all processes and since the sending process is already blocked (REPLY).

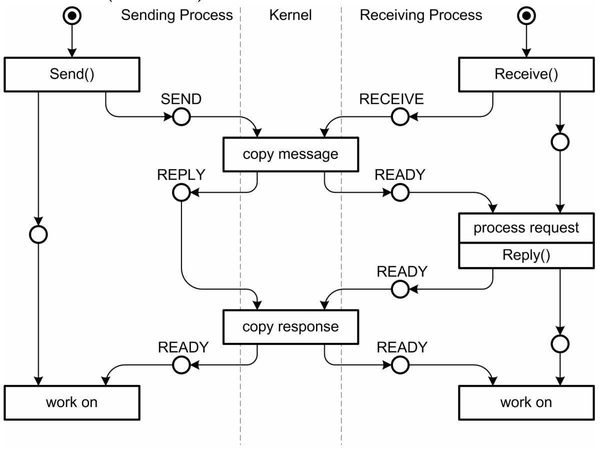


Figure 6 Message passing in QNX.

Obviously, this indirect communication results in a lot of overhead. For a single service request, two messages have to be copied from one address spaces into another one and multiple context switches occur. On the other hand, the complete protection of address spaces ensures that no process can read or write data belonging to another process. This certainly helps developing reliable and secure systems.

3.2. The OSA+ middleware platform

OSA+ is a middleware platform that can be scaled to run on hardware ranging from DSPs[1] and microcontrollers to PCs and workstations. The OSA+ platform enables services running on the same or different computers to interact in a transparent way by using so-called jobs for communicating with each other.

The OSA+ core contains only basic functionality for service and job management. In its minimal configuration the OSA+ core offers only local jobs and services (i.e. no network communication), a limited number and size of jobs and services, strictly sequential job management (i.e. no multitasking), and no real-time monitoring.

If further functionality is required, and if the target platform has enough resources, basic and extension services can be used to scale the OSA+ core to the specific needs of a system. These services are plugged into the core in the same way as user services are.

Basic services remove the above-mentioned restrictions from the system. A memory service gives access to memory management, allowing more and larger jobs and services, a process service allows multitasking, a communication service provides access to remote services, and an event service permits the monitoring of real-time constraints by managing timer and other hardware events.

The purpose of extension services is to add further functionality to the OSA+ core such as event logging and security services.

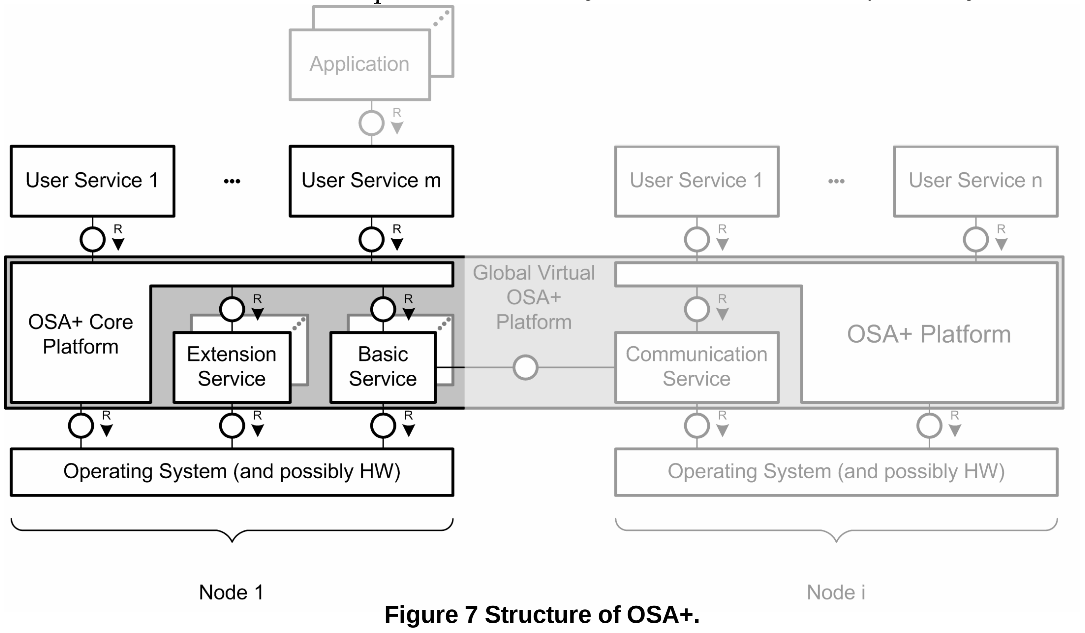


Figure 7 Structure of OSA+.

[1] DSP: Digital Signal Processor

3.3. The MicroKernel Database Engine

The MicroKernel Database Engine is the core of Pervasive.SQL V8. While the MKDE implements the internal layer of a database management system, plug-in modules implement the conceptual layer, providing for example transactional or relational data access modes.

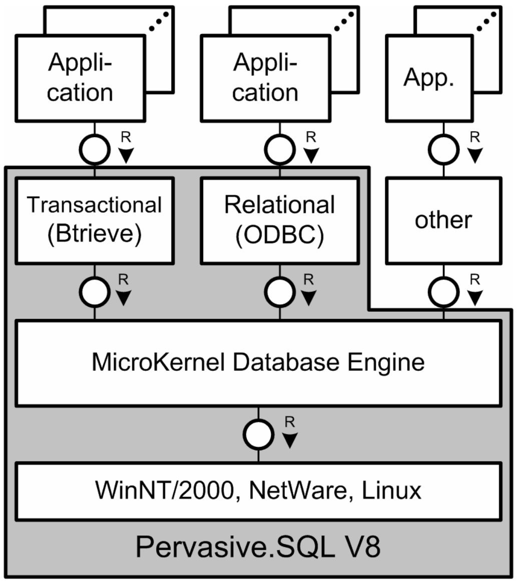


Figure 8 Structure of the MKDE and Pervasive.SQL V8

The MKDE provides low-level data management services such as physical data access, transaction processing, durability, data and referential integrity enforcement, data caching and event logging.

The plug-in modules may provide further functionality in order to implement their respective data access modes. For example, the relational access module provides atomic statements, bidirectional cursors, outer joins, updateable views, ODBC data types, triggers, stored procedures and security.

In the context of the microkernel architecture pattern the plug-in modules are system views offering different abstractions of the physical data managed by the MKDE. Applications can access these modules by using appropriate drivers/adapters (e.g. an ODBC driver). [PERVPROD]

4. Related Patterns

As pointed out in [POSA1996], the microkernel architecture pattern could be considered as a special case of the *layers pattern*. The microkernel and the system services would constitute the lowest layers, offering a basic abstraction of the underlying hard- or software platform. System views and adapters would be further layers. Moreover, the *layers pattern* could proof to be useful for implementing many of the components involved in a microkernel system.

The *reflection pattern* could be used to enable a microkernel system to be configured and changed at run-time. In such a system, components could be dynamically loaded and unloaded, enabling the system to respond to a changing environment without restart.

The *broker pattern* can be used to develop distributed microkernel systems. In such a system the microkernel contains a network communication interface, enabling system components that run on different machines to interact in a transparent manner as if they were running on the same machine. QNX and Amoeba [DistSys91] are examples of distributed microkernel operating systems. OSA+ on the other hand could be regarded as a *broker system* based on a microkernel architecture.

For implementing those components running as separate processes, multithreading patterns such as *leader-followers* could be used for achieving an efficient handling of synchronous and asynchronous requests. Furthermore, the *proxy pattern* could be used in adapters in order to optimize component interaction by caching responses to certain requests.

5. Concluding Remarks

As the examples have shown, the microkernel architecture pattern can be applied to many different application domains – not only to operating systems. Usually, these systems are application platforms for third-party applications or modules. Thus, it is important that they provide a reliable environment, protecting the whole system against possibly malicious or faulty applications or modules. Other examples of application domains than those already mentioned could be: web servers, applications offering plug-in or scripting support (such as web browsers and editors), or software systems that need a distribution, communication, or resource management mechanism in need of more rights than most other parts of the system.

Depending on the application domain and on the constraints and requirements guiding the system architecture, this pattern can be realized in various ways. However, some characteristics are common to all microkernel systems:

- The microkernel comprises only a minimal function set. This makes the microkernel as small as possible, fostering maintainability and changeability.
- A microkernel system can be easily scaled and adapted by adding or removing system services as needed.
- All other system components depend on the microkernel – directly or indirectly.
- The microkernel, serving as an abstraction layer, makes porting of applications easy. In general, only the underlying microkernel system has to be adapted.

5.1. Benefits

The microkernel architecture pattern is very useful for developing software systems that have the above-mentioned qualities (ending in –ility): portability, maintainability, changeability, configurability, extensibility, reliability, scalability … Further benefits are:

- Clearly structured system design: This pattern promotes the use of many independent system components which have well-defined tasks. Such systems are generally easier to understand, to maintain and to adapt to changing requirements.
- The distributed microkernel variant may yield further advantages such as fault tolerance, increased availability and transparency (i.e. inter-process communication over a network is transparent to processes).

5.2. Disadvantages

Despite all these benefits, using the microkernel architecture pattern may also have some disadvantages:

- The process of designing a microkernel system is a very complex task. It requires profound knowledge of the application domain. Especially for small projects, using this pattern may result in a too large design overhead.
- As has been pointed out several times, using many separate processes generally leads to an increased overhead which may in turn decrease overall performance.
- In some cases, this pattern may complicate the implementation of certain functionality. Where otherwise a simple procedure call might suffice, complex inter-process communication may be needed in a microkernel system.

5.3. Limitations

Of course, the microkernel architecture pattern is no magic bullet. It does not guarantee portability and changeability (and all other -ilities) in every case.

- Sometimes, the microkernel may need to be changed when additional system services are added. Especially, if these services implement completely new functionality the API of the microkernel may have to be adapted in order to accommodate new system calls.
- In [WIN2000] it is pointed out that a crash of an important system service may lead to a crash of the whole microkernel (operating) system.

When considering the microkernel pattern, the main difficulty the system architect has to pay attention to is the rather complex task of designing a proper microkernel system which *can* in turn lead to an easy-to-understand system. Using the microkernel pattern does not automatically guarantee the above mentioned benefits.

Simple applications that are not intended to be portable to all kinds of platforms or that do not have to be extensible should avoid the design overhead. These systems are typically built for a limited field of application or a small group of customers (in contrast to standard software). If time to market is more crucial than for example maintainability or if performance is of utmost importance (particularly where processing resources are short), applying the microkernel pattern might be a bad choice.

5.4. Modeling

The microkernel pattern is a great example of how using FMC increases overall system understanding. Firstly, FMC focuses on system structure and behavior rather than on illustrating the exact representation of processes, classes, devices etc. This allows recognizing patterns more easily, as pointed out by figures 2, 5, 7 and 8, which can be mapped on figure 1 without great efforts. Additionally, FMC makes it possible to show the different aspects of a system design. In contrast to figure 5 which reveals the microkernel pattern, figure 4 concentrates on the abstraction layer "processes" and thus demonstrates the commonness of all processes in respect to the microkernel. It is important to emphasize FMC's semiformal, nearly informal approach that improves inter-human communication about systems. This approach cannot (and even does not want to) replace traditional, code oriented modeling techniques but it is extremely useful for supplementing them.

References

[DistSys91] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, Andrew S. Tanenbaum, A Comparison of Two Distributed Systems: Amoeba and Sprite, 1991

[FOLDOC] Free On-Line Dictionary of Computing, http://www.foldoc.org

[OSA+2000] U. Brinkschulte, C. Krakowski, J. Riemschneider, J. Kreuzinger, M. Pfeffer, T. Ungerer, A Microkernel Architecture for a Highly Scalable Real-Time Middleware, 2000

[PERVPROD] Pervasive.SQL V8, Pervasive Products and Services, Pervasive Software Inc., 2003 (?), http://www.pervasive.com/support/technical/psqlv8/prodserv.pdf

[POSA1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stahl, Pattern-Oriented Software Architecture – A System of Patterns, Wiley 1996

[QNX] QNX Neutrino RTOS, http://www.qnx.com

[TanVsTor1992] Open Sources: Voices from the Open Source Revolution, Appendix A – The Tanenbaum-Torvalds Debate, Andrew S. Tanenbaum, Linus Benedict Torvalds et al., http://www.oreilly.com/catalog/opensources/book/appa.html

[VRTX] VRTX Real-Time Operating System, http://www.mentor.com/vrtxos/

[WIN2000] David A. Solomon, Mark Russinovich, Inside Microsoft Windows 2000, 3. (German) edition, Microsoft Press 2001

Evaluating and Extending the Component Configurator Pattern

Stefan Röck, Alexander Gierak
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{stefan.roeck, alexander.gierak}@hpi.uni-potsdam.de

Abstract

The Component Configurator Pattern is used when an application needs to dynamically link and unlink its components at run-time without being recompiled or statically relinked.

We will provide an overview and present important aspects of the pattern. Then we use the Fundamental Modeling Concepts (FMC) to show an alternative possibility of the modeling and propose extensions, for example with the Observer pattern, to increase functionality and applicability. Furthermore examples are given succeeded by an evaluation.

Keywords: Component Configurator, Patterns, Architecture, Configuration

1. Introduction

Design patterns are well known in the software developing community and used for many years in building software systems [Gamma1994]. As participants of the “Conceptual Architecture Patterns” Seminar at the Hasso-Plattner-Institute in 2003 we studied different common design and architectural patterns.

In this paper we will focus on the *Component Configurator* pattern. We will start with descriptions of the pattern and which possibilities are there today to model it. Then we will present our own modeling results with the Fundamental Modeling Concepts [FMC]. We also propose some suitable extensions in order to enhance functionality and applicability. Afterwards we will discuss the relation to other patterns, evaluate the usability of the pattern and elicit some uncertainties. Finally we will discuss the questions whether the pattern could be an architectural pattern with our proposed extensions.

2. Pattern description in POSA

2.1. Configuration

In an encyclopedia configuration is described as “Something (as a figure, contour, pattern, or apparatus) that results from a particular arrangement of parts or components. “ [Merriam-Webster]

Related to Software it is very important to ask when configuration takes place and how it is implemented. The basic difference is whether the application is running or not during configuration. Changing the configuration at run-time is in most cases more complicated. For this pattern changing the configuration is accomplished in two ways. First by changing the implementation of a component and second by reinitializing a component with new parameters. Both aspects are addressed by this pattern.

2.2. Definition

For our research we focused on the book “Pattern-Oriented Software Architecture” by D. Schmidt, M. Stal, H. Rohnert and F. Buschmann [POSA2000].

In this book the Component Configurator pattern is defined as followed: “The Component Configurator design pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application. Component Configurator further supports the reconfiguration of components into different application processes without having to shut down a re-start running processes.”

2.3. Context and Requirements

The Component Configurator pattern is suitable for applications or systems, in which components must be initiated, suspended, resumed and terminated as flexible and transparent as possible.

Hence there is a clear need for a mechanism to configure the components into an application which meets the following requirements:

- Ability to modify component implementations at any point during an application's development and deployment lifecycle
- Modifications to one component should have minimal (ideally none) impact on the implementations of other components in use
- Ability to initiate, suspend, resume, terminate or exchange a component dynamically at runtime
- Administrative tasks should be straightforward and component-independent

2.4. Participants

There are four participants of the Component Configurator pattern described in [POSA2000]:

- The *Component* defines a uniform interface for configuring and controlling a particular type of application service.
- The *Concrete Component* implements this interface.
- The *Component Configurator* controls the linking and unlinking of concrete components into and out of an application.
- The *Component Repository* is used by the Component Configurator to manage all concrete components configured into the application.

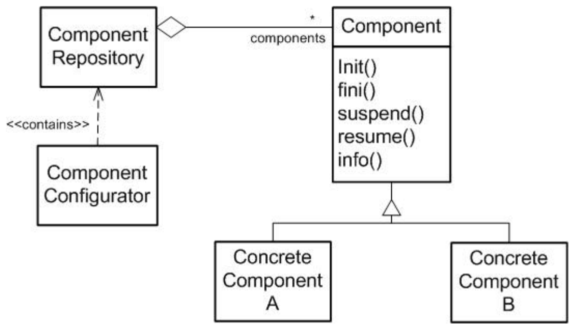


Figure 1 Class Diagram [POSA2000]

The relation of the participants is described in figure 1 as an UML class diagram.

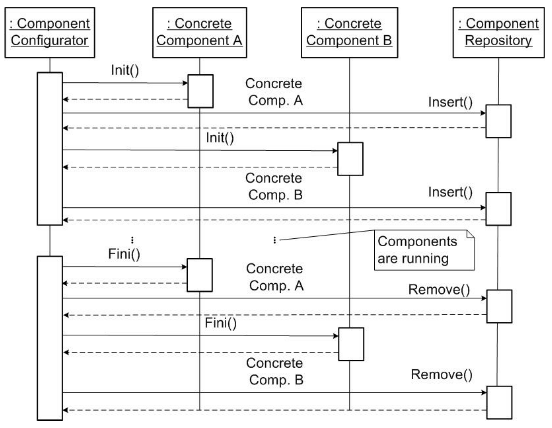


Figure 2 Sequence Diagram [POSA2000]

As shown in the class diagram the component interface has five specific procedures which are used to uniformly configure the concrete components. An example control flow is described in figure 2 (UML sequence diagram). In this example, two concrete components are inserted into the component repository and then removed.

3. System view of the pattern

One problem we faced is that the pattern alone is not much more than an interface definition with five procedures, but the basic aspect of this pattern, the dynamic reconfiguration of a component, is not completely pointed out. In order to go beyond it is necessary to embed the pattern in a surrounding system, to create an environment in which the Component Configurator could possibly work. At first we choose a basic scenario and then develop it to a higher complexity which goes along with extensions of the pattern as well.

3.1. First Approach

The modeling of the Component Configurator in [POSA2000] does not satisfy our needs, thus we modeled the pattern with the Fundamental Modeling Concepts (FMC) to get a clear understanding of the pattern's main characteristics. Therefore we need to make several assumptions of how the pattern's participants might integrate within a system.

3.1.1. Compositional Structure

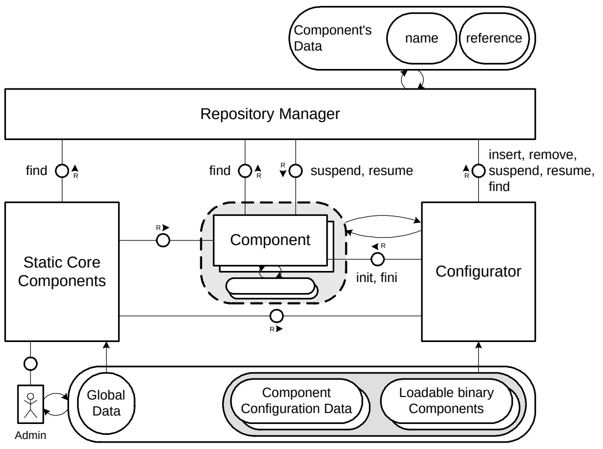


Figure 3 Structure of the Component Configurator (block diagram)

A FMC block diagram uses passive and active components. Active components are called agents which communicate with each other using passive components (channels or storages). A higher level of abstraction allows us to focus on the relevant participants within the system and show only their important relations. Consequently, central agents and their relations showed in figure 3 need not to be identical to objects or classes in an implementation of the pattern.

A central role plays the agent Configurator which is responsible for loading and configuring dynamically loadable components, e.g. DLLs, as well as for their reconfiguration and destruction. This procedure is triggered by an agent named Static Core Components which represents all parts of an environmental static system. The configuration data are stored in a global storage which can be modified for instance by an administrator.

In figure 3, the storage with the dashed line holding components shows a structure variation which underlines that creation and destruction of components may take place at any time. The Repository Manager encloses all components by mapping of a component's name to its actual reference (which is transient e.g. due to substitution). The mapping is done by using the find()-procedure. The Repository Manager has been introduced to clearly separate the task of holding references of components.

Again we make some assumptions to retrieve a clear understanding of suspension and resumption of components. In [POSA2000] the responsibility for these issues is not clearly pointed out. We propose that the Configurator must not directly suspend and resume components but uses the Repository Manager to do so. This implies that he does not need to store component's references but can simply pass the component's name (received, for instance, from the Static Core Components) to the Repository Manager which can perform the mapping and the operation on the desired component.

Otherwise we would have to hold two tables with names and references or would have to implement are shared memory for Configurator and Repository Manager. Both solutions would imply an unnecessarily more complex system due to problems of consistency or separation of concerns.

3.1.2. Dynamic Structures

An administrator puts binary component and configuration data into a global storage and forces the Configurator via a Static Core Component to process these files. After creation and initialization an insertion request is passed to the Repository Manager. This agent makes the component public to all other participants in the system by keeping the (unique) name and a reference (e.g. a pointer or IP-address with port number) in memory. Only as from now the component is inserted into the whole system.

At this point the question arises who actually is responsible for the creation of components. The answer is that it depends on the level of abstraction. On a very abstract point of view the Repository Manager takes this role for reasons mentioned in the paragraph above. Regarding a rather technical view the Configurator creates components although no one except himself can take advantage of the component's functionality.

The procedure of reconfiguration looks quite similar to the insertion except the suspension and resumption of components. In [POSA2000] these issues are not described in detail: in fact, it is only said that it should be possible to suspend and resume a component but it is not mentioned by whom and when. A solution might be to suspend all dependable components during the reconfiguration process only. A scenario which manages suspension and resumption indirectly is described in detail in the following sections.

3.2. Extension I – Clarifying semantics

As pointed out in the section above the aspect of suspending and resuming components is described only superficially. Therefore we will now suggest some extensions with the goal to make the pattern easily and generically applicable.

An obvious question is which components should be suspended how long in which scenario. We propose to extend the Repository Manager's functionality in a way that additionally to a component's name and reference the information about connected components is stored. This can be easily achieved if the requirement is met that all participants who want to start a communication with a component call the find()-procedure. Furthermore a mechanism for signaling the end of a communication process is needed. That's why we introduce another procedure closeConnection() (see figure 4) which provides this desired functionality. Both procedures allow the Repository Manager to keep track of all connected components for each dynamically loadable component.

After a reconfiguration request occurred the Configurator asks the Repository Manager to remove that component. This is done by waiting for the end of all open connections and by setting the allowConnection flag to false which indicates that no new connections will be allowed. The Repository Manager can manage this by returning an error value on new find() requests. After all connections are closed the remove() procedure returns and the actual reconfiguration process can start. Later on the new component is reinserted into the Repository Manager.

This proposal provides a solution for problems which might occur if a component in reconfiguration process receives requests which it cannot handle at the moment. Additionally a component can be safely removed without disrupting existing connections which might result in unpredictable behavior. Nevertheless the suspend() and resume() procedures as proposed in the original pattern description become obsolete as this functionality can be considered to be implemented within the Repository Manager agent. In fact, an exchange of roles takes place because not the Configurator but the Repository Manager allows suspension and resumption by enabling / disabling connections between two components. Nevertheless the Configurator still triggers these actions by removing and re-inserting an existing component.

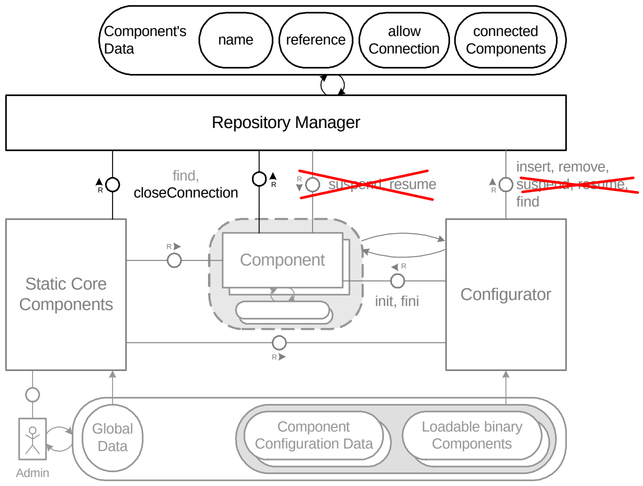


Figure 4 Static Structure Extension I

3.3. Extension II – Notifying observers

The suggestion provided in section 3.2 has (at least) one big disadvantage: no one except the Configurator knows how long a configuration process lasts and until when the affected component is unavailable. This is not only inefficient but could also lead to strong lack of performance within the whole system. Imagine an agent which provides weather information to clients. During reconfiguration (J. Kachelmann opened the 65.537th weather station which makes an adaptation of the database connection necessary) clients periodically send requests to the agent and receive "Agent currently not available" responses. This cause a lot of traffic and due to lower bandwidth could affect other applications on the network.

Consequently we suggest to combine the Observer pattern as described in [Gamma1994] with our proposal from section 3.2. The Configurator plays the subject, static and dynamic components the object role. As shown in static structure in figure 5 components register for each component they are going to communicate with (at a later time). The Configurator keeps a list of dependables for each component and notifies via update() all registered participants that a concrete component is currently unavailable. If the reconfiguration process is finished a new notification is sent. All components themselves keep track of all notifications they receive so that they can determine at any time which components are "on air" and which are not.

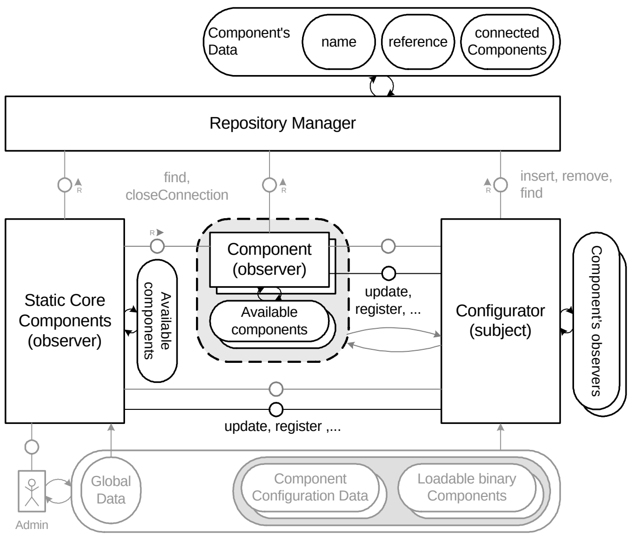


Figure 5 Static Structure Extension II

A side effect of using this solution is the absence of the allowConnection-flag in the Repository Manager's storage which no longer is necessary as all components store this kind of information themselves. Although there is overhead due to registration and notification mechanisms fewer messages might be necessary if clients from the example above are considered to send only one request for retrieving weather information because they know which service component is available at the moment. They also possibly can send their requests to another redundant component which provides better response times.

One might think of further adaptations, e.g. of a Repository Manager that stores information about performance issues and automatically selects and returns a reference to the most suitable component. However, this is beyond the scope of this paper.

Of course, this approach is adaptable for relatively small systems with few components only. Otherwise the need of much memory and the amount of control messages become problematic.

3.4. Memento Pattern

An important issue when substituting components is the restoration of the old state. If the requirement exists that a new component must continue exactly at the point where the old component stopped the memento pattern as described in [Gamma1994] is applicable. In the extensions we proposed the Configurator can take the responsibility for preserving the state before calling the fini() procedure. After the init() procedure is processed the state can be transferred back.

4. Examples and related patterns

4.1. Java Applets

The principle of Java applets is an example for the use of this pattern which is often depicted in literature. The main aspects initialization, suspension and resumption and the process of dynamic loading can be found here. The interface which an applet provides is quite similar to the description in [POSA2000] with the exception that no possibility for termination is available. As seen in figure 6 the role of the Configurator is distributed among different agents that can be found in different locations of the system as well.

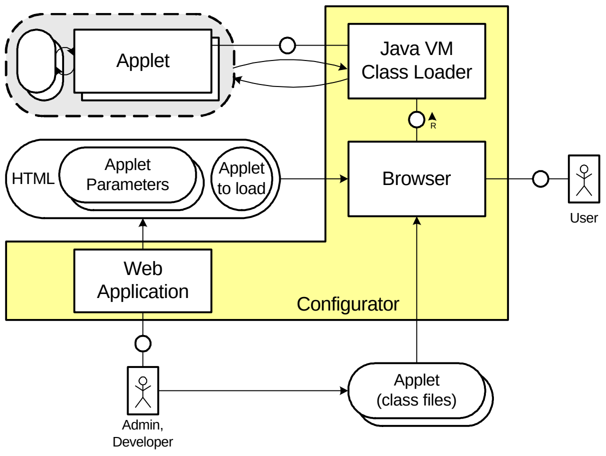


Figure 6 Compositional Structure Java Applets

The Java Virtual Machine (VM) together with the browser is responsible for loading an applet. The configuration parameters are retrieved from the web. The corresponding agent decides whether an applet is part of the system and can substitute applets by giving a different reference within an HTML-document – in so far he is also part of the Configurator agent.

In this example we can see that the roles of configuration (Web application) on the one side and of management of components (Browser & Java VM) on the other side are clearly split which is not the case in the proposals we made above.

4.2. Apache Web Server

The dynamically loadable components within the Apache Web Server are modules which have different tasks and which provide a good possibility for extending the functionality. Apache modules match in so far with the properties of the pattern that they can be (re-)configured during runtime (via graceful restart) and a special agent is responsible for loading and management issues (mod_so, which is itself a module) [Apache].

Furthermore, Apache modules meet the requirement that they have a common interface (register_hooks procedure). However, they use the mechanism of hooks (procedures are first registered at Static Core Components and later on called by them) which provides a very dynamic handling. Neither suspension nor resumption of modules is possible. The Repository Manager is implemented as a global data structure (pool) where information about modules, their hooks and cleanups are stored. Cleanups are necessary to make sure that the module is unloaded correctly in the case of a server shutdown. In fact, these are procedures which are called by Apache core – that's why it has modifying access to the structure variation enclosing the modules in figure 7.

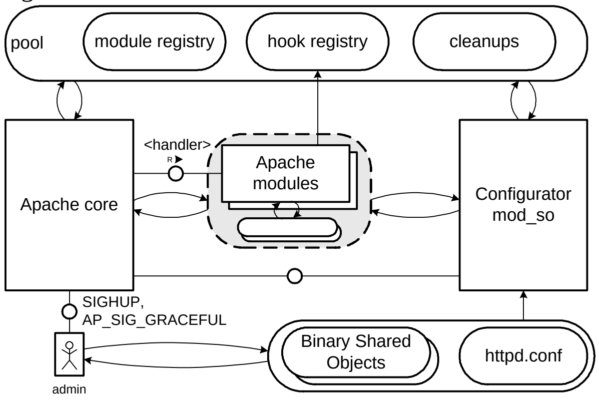


Figure 7 Compositional Structure Apache Modules

4.3. Related Patterns

In most cases design or architectural patterns are not used isolated but are connected with each other. The Component Configurator pattern is typically used by the architectural patterns described in the next sections.

4.3.1. Pipes and Filters

As the names states this pattern uses filters connected by pipes to efficiently process data. Such a sequence of (generally) independent, but adjacent processing steps is called a filter chain. When the application needs the ability to dynamically create filter chains at run-time, for example if different input sources exist, then the Component Configurator pattern can be used.

4.3.2. Broker

The broker pattern is a communication pattern used when components in a distributed environment should be able to communicate as if they were in a non-distributed environment. To accomplish this goal local broker agents and proxies [Gamma1994] are used. Typically there is a need to exchange or migrate these components at run-time. This can be done with the use of the Component Configurator pattern.

5. Conclusions

5.1. Evaluating the pattern description

In our opinion the description of the Component Configurator pattern in [POSA2000] is not sufficient. The structure of the pattern and the scenarios when applicable are described quite well. However, for concrete questions like how the exchange of a component is handled in detail no answers are provided. Especially control flow structures are missing and concerning that just an idea is stated in the book but not a solution. Furthermore, the pattern is described close to the implementation point of view while we think a higher level of abstraction would be more helpful.

The modeling of the pattern is limited to a class and a sequence diagram (figures 1 and 2) and therefore not satisfying for a lot of imaginable scenarios. We present our modeling results with FMC and propose extensions to the pattern which allow to understand and, even more important, to control the configuration process clearer and increase quality of service characteristics like performance or availability by using the observer and memento pattern.

Furthermore the question if the Component Configurator is a design or an architectural pattern arises. Architectural patterns define fundamental structural organization schemes for software systems and provide a set of predefined subsystems. Design patterns provide a scheme for refining subsystems or components of subsystems, or the relationship between them [POSA2000]. Although we know that no unambiguous definition and clear dividing line between those expressions exists we think that the Component Configurator with our extensions and modeling is an architectural pattern. It provides a solution for a concrete problem in a quite abstract manner and can be adapted to numerous scenarios as the given examples prove.

5.2. Other aspects and final remarks

A point which has not been considered yet is the type of operating system in which the pattern is used. For instance, it makes a large difference when thinking about suspension and resumption if all components (static and dynamic) are running within one thread, if each has one or more threads or if they even run in different processes. This example again illustrates the problems with the original pattern description as direct communication between classes (i.e. function calls) does not work between different processes. Also possibilities of interactions (shared memory, remote procedure calls, etc.) between components depend on the concurrency model.

Another aspect is the distribution of components which is not excluded in our modeling, however new aspects as balancing, fault tolerance and more sophisticated communication mechanisms, provided for instance by a middleware platform, play an important role.

After all a generic evaluation cannot clearly been drawn. As the pure pattern description comprises only few facts it can be adapted to nearly every scenario where configuration plays a role. However the software architect has to reflect the current requirements and cannot rely on a completed solution in that way as with other patterns. However, we tried to propose some generic scenarios and showed how this pattern can be used.

References

[Gamma1994] Gamma, E. et. al. Design Patterns, Addison Wesley, 1994

[POSA2000] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-Oriented Software Architecture. Wiley, 2000

[Merriam-Webster] www.m-w.com / home.htm

[Apache] apache.hpi.uni-potsdam.de

[FMC] www.fmc.hpi.uni-potsdam.de

The POSA Interceptor Pattern

Marc Förster, Peter Aschenbrenner
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{marc.foerster, peter.aschenbrenner}@student.hpi.uni-potsdam.de

Abstract

The Interceptor architectural software pattern described in [POSA00] offers a mechanism to extend software systems by adding new services, without affecting the existing system's static or dynamic structures.

In this paper we present a digest of the pattern in general, look at the problems it is meant to solve, and show practical examples of application. Throughout we will use the system modeling technique FMC for diagrams.

Keywords: Software Architecture Pattern, Interceptor Pattern, Software System Extensibility

1. Introduction

Interceptor is a variant of the Chain of Responsibility behavioural pattern [GOF95], decoupling communication between the sender of a request and its receiver. It enhances the flexibility and extensibility of a software system by letting applications add to the base system's functionality and also dynamically change its subsequent behaviour.

In contrast to [POSA00] we use the [FMC] system modeling technique to represent the runtime structures (static as well as dynamic) of the Interceptor pattern.

2. The Problem

In general, not all of the functionality a system will have to offer in the future can be anticipated during development. A web browser, e g, should be able to display images in formats that will evolve over time, or that do not even exist today; a server farm should allow users to add a load balancing facility of their own choosing to the system. Putting in too much functionality would render the system huge and lead to unneccessary overhead, both in system development and at runtime.

To seamlessly integrate such applications in a system they should be able to monitor and manipulate its behaviour. At the same time this should not require changes in the design or implementation of the base system.

Furthermore, because stopping and recompiling a system to include new services is not always possible, it is desirable that additional services can be included at runtime.

3. The Solution

3.1. Requirements

To solve the problems mentioned it is necessary to introduce a mechanism for extending a software system's functionality by

- registering new services with the system,
- letting the system trigger these services automatically when certain events occur, and (optionally)
- letting the new services access the system's internal state and control its behaviour.

By applying the Interceptor pattern all of these issues can be addressed.

3.2. The Interceptor Pattern

3.2.1. Compositional System Structure

The Interceptor pattern involves two collaborating main agents: an application and the base system it is meant to extend. Communication between application and base system core is mediated by three types of components: dispatchers and contexts (belonging to the base system), and interceptors (being part of an application that extends the base system's functionality). These components form a kind of abstraction layer:

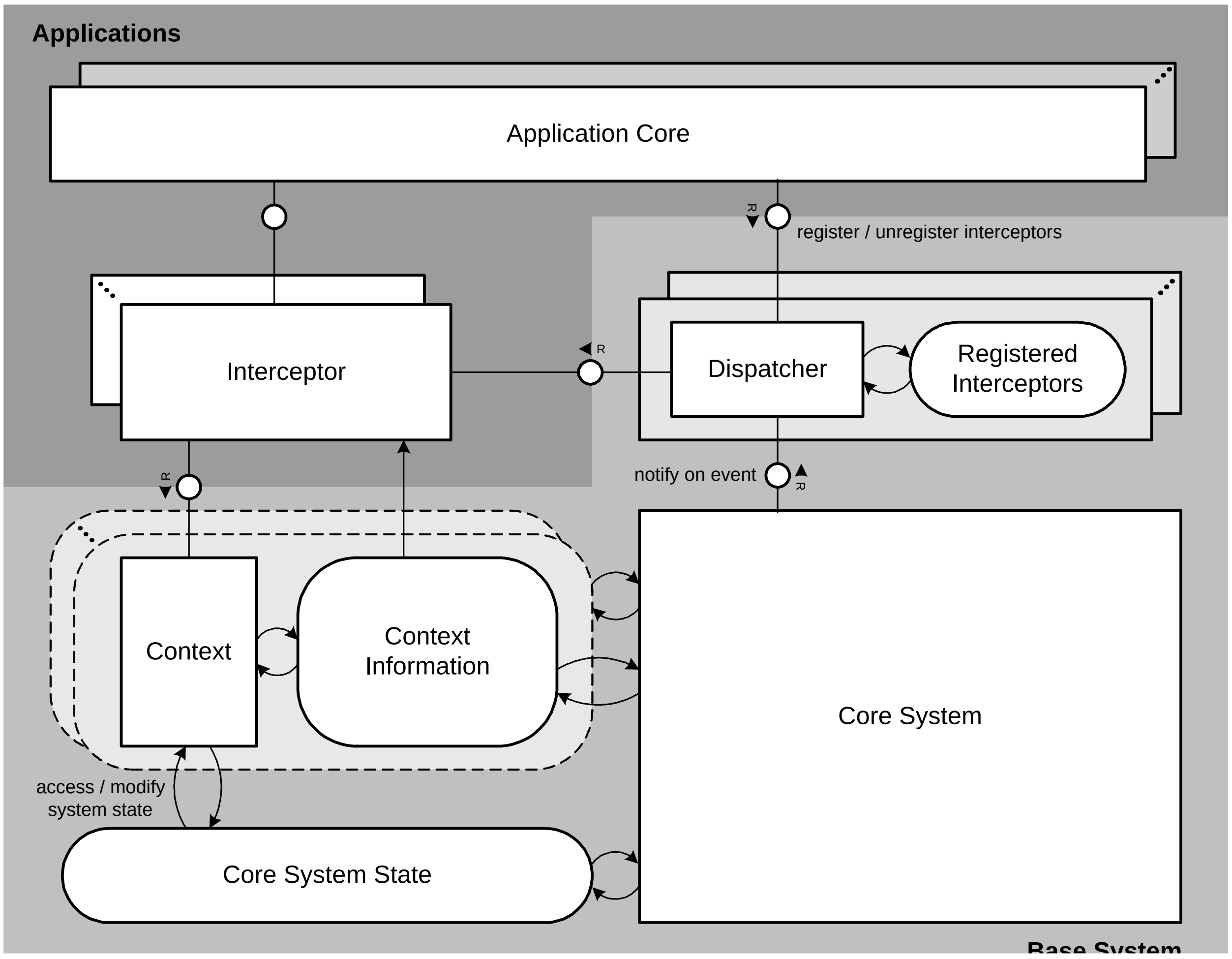


Figure 1: Basic compositional system structure of the Interceptor pattern

dealing with an event is delegated by the dispatchers to interceptors, confining adaptation to different base systems to a small part of the application. Figure 1 shows the compositional structure of the pattern, with the participating agents and their communication channels (interfaces).

3.2.2. Participating Agents and their Interfaces

The *core system* contains all basic system functionality that is made extendable by external applications. Whenever one of a set of predefined events is triggered (e g, a web server receives an HTTP request) the core system notifies its dispatchers. Besides that it creates context agents containing information read by interceptors and passes a reference to them to the dispatchers. This can either be made to happen per event triggered or once for each interceptor registration.

The *dispatchers* offer to the *application* an interface for the registration of its interceptors, i e, additional services. Thus, the core system does not need to know anything about the application or the services performed by it. When a dispatcher gets notified on an event it iterates its registered interceptors, calling the appropriate interceptor routine, and passes on the context reference given to it by the core system (in the case of per-registration strategy the interceptors already know the corresponding context). Typically there is one dispatcher for each interceptor.

The *contexts* contain information on the concrete event that has been triggered (in the per-event creation strategy, see above) or, more generally, on an event type (in the per-registration strategy). This information is used by the interceptors to process the event. In case interceptors do not need to manipulate the core system, the context can be just a passive storage location. In case interceptors are granted modifying access to system state in order to change subsequent sys-

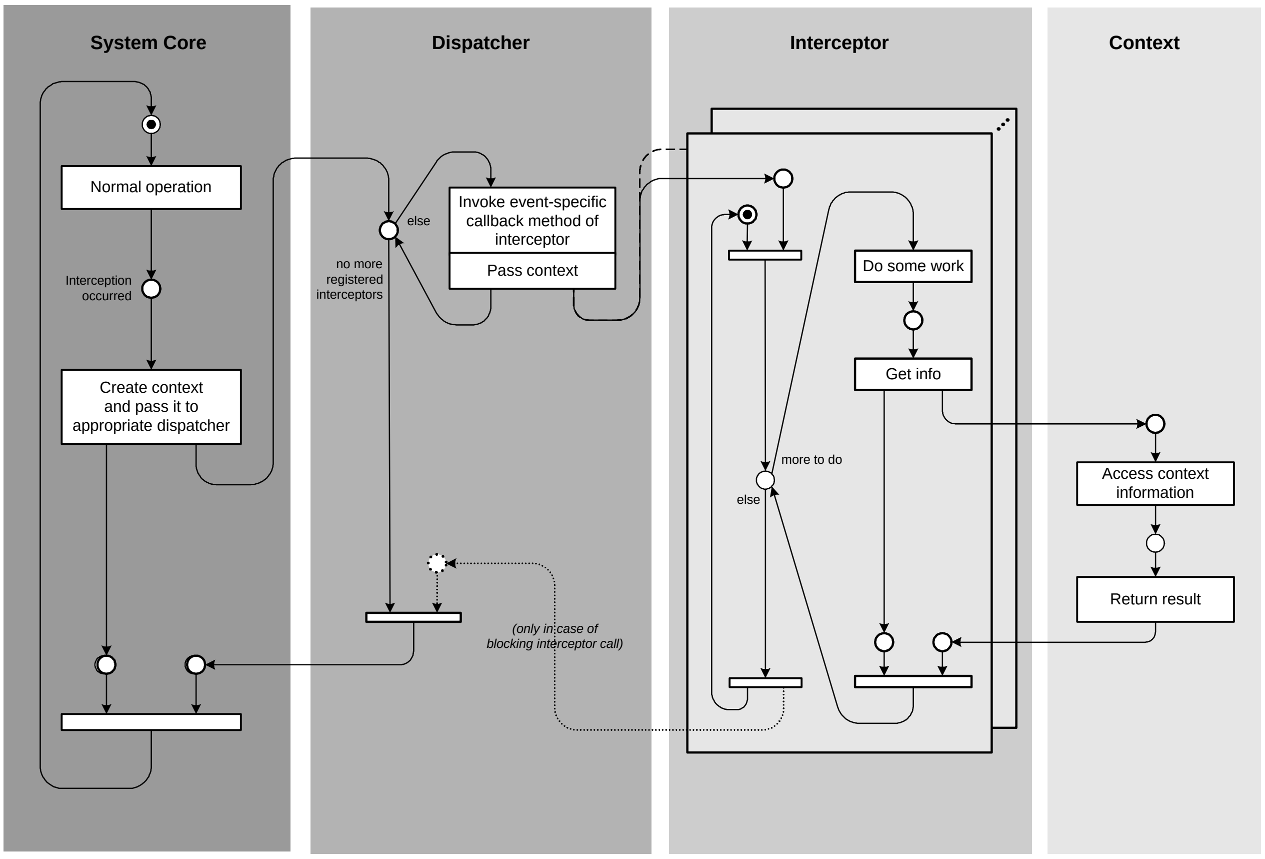

System Core
Dispatcher
Interceptor
Context
Normal operation
Interception occurred
Create context and pass it to appropriate dispatcher
else
no more registered interceptors
Invoke event-specific callback method of interceptor
Pass context
(only in case of blocking interceptor call)
Do some work
Get info
more to do
else
Access context information
Return result

Figure 2: Basic dynamic system structure of the Interceptor pattern

tem behaviour, the context will be an active system component (depicted in Figure 1) mediating and thereby limiting such, potentially harmful, access.

The *interceptors* pass the service request from the dispatcher to the application. From the given context they read the information needed to process the event. Optionally, an interceptor can influence the subsequent behaviour of the core system (i e, change core system state) through the channel to its associated context actor, e g, for a load balancing service that needs to redirect incoming requests to the least busy server.

3.2.3. Dynamic Structure

After the application has created its interceptors and registered them with their dispatchers and an event happens in the core system that is to be processed externally

1. the system core notifies a dispatcher of the occurrence of an event. Optionally it passes a context (reference) along, if the per-event strategy is set.
2. the dispatcher notifies its registered interceptors of the event. Optionally it may block and wait for a reply but that is not always necessary, e g, for a system monitoring tool.
3. the interceptors perform their tasks and use the contexts to obtain information and / or change the base system's subsequent actions.
4. the base system continues after the dispatcher has returned. In case the dispatcher blocks waiting it will wait, too.

Figure 2 shows the dynamics triggered by an event, when the per-event strategy for contexts is chosen, the interceptor call is either blocking or non-blocking, and the interceptors cannot modify the core system's state.

3.3. Design Activities

When applying the interceptor pattern to a concrete software design several steps are necessary in order to define specific details of the pattern implementation. These steps are called design activities.

3.3.1. Identify interception points

The first step is to identify the interception points within the core system. This is to identify the points of time in the system's dynamic behavior in which possible interceptors can be triggered.

A preferred way to do so is to model the dynamic behavior of the system. An appropriate model helps to determine not only which interception points exist, but also aids in grouping them. Appropriate models for this task are petri nets and state machine diagrams.

Interception points can be grouped in two ways. One is to distinguish between the possible behaviors of the corresponding interceptors with the core system. The interceptors can have read- or write- access to the system and grouping the interception points in reader- and writer- points makes it easier in later design activities to specify the interceptors.

Forming interception groups is the second way to group interceptors. Semantically similar interception points are combined into one interception group. This helps in minimizing the amount of required dispatchers as only one dispatcher per interception group is needed and not one per every interception point. Semantically similar can be several things. One possibility is to group interception points that deal with the same issue in the system, e.g. all interception points that deal with the sending of the response in a web server.

3.3.2. Specify contexts

The next design activity is to specify the contexts which are used to retrieve information from and modify the behaviour of the core system.

First of all the grouping of interception points into reader- and writer- interception points from the proceeding activity helps to determine how the context for a specific interceptor has to look like.

If the interceptor has solely read access to the system the context doesn't have to provide an interface that allows the modification of the system but it has to provide the information that the interceptor expects. Interceptors with write access require both, yet the context defines which aspects of the system can be modified.

Another decision regarding the interface of contexts can be made to optimize the number of needed contexts.

The multiple interfaces strategy means that for every type of interceptor a different context has to exist, as a specific interface and therefore context is specified for each type of interceptor.

In contrary the single interface strategy depicts the fact that only one interface and therefore only one context exists for all interceptor types. Due to their nature a single interface can become very broad and unhandy while multiple interfaces may result in too many contexts in use. The actual strategy used can be a mix of single and multiple interfaces and the exact amount of contexts and therefore interfaces has to be balanced out for every project.

The last thing to specify is the way contexts are created during the processing of the core system. Again two different strategies can be applied here, per-registration and per-event.

In the per-registration strategy the context is only created once when a dispatcher is registered while a new context is created for every invocation of a dispatcher in the per-event strategy. The information provided by the context in the per-event strategy can be event specific, one the other hand the constant creation of contexts can lead to a big overhead. The per-registration strategy is the exact opposite. It has no overhead but the information provided by the context can only be very general.

3.3.3. Specify interceptors

The main task in the design step of specifying the interceptors consists in defining the interceptor interface. It is used by the dispatcher to trigger the interceptor and to pass the context.

The dispatcher provides a set of callback methods, which registered interceptors can implement. So the definition of the interface for the interceptor invocation has to be conform to the methods defined in the dispatcher.

The passing of the context can be done in two ways, either by passing the context itself or by passing a reference to the context.

3.3.4. Specify dispatchers

The last design activity is to specify the dispatchers. This activity includes the definition of two interfaces, one for interceptor registration and removal and one for the notification of the dispatcher by the core system and the definition of the interceptor invocation strategy.

The interface for interceptor registration and removal is used by the application core which wants it's interceptors to be registered or removed from the dispatcher. This interface depends on how the dispatcher invokes it's registered interceptors. If the dispatcher for example uses a priority based invocation strategy, the priority of the interceptor has to be given to the dispatcher via the registration interface from application core.

The interface for the notification of the dispatcher from the core system is very similar to the interface for the invocation of interceptors from the dispatcher. The core system defines callback methods which are implemented by the dispatcher. If the methods defined by the core system to trigger the dispatcher and the methods defined by the dispatcher to trigger the interceptors are identical, the dispatcher doesn't even have to implement the methods, as it's only function is then to forward the method call to the appropriate dispatcher.

As already mentioned earlier an invocation strategy for the interceptors has to be defined too. There are different ways to implement such a strategy ranging from a first come first serve to a priority based or a dynamically configurable strategy.

3.4. Application Examples

The typical range of application for the interface pattern is software systems that should be highly extensible. A good example for that are the middleware systems CORBA and COM as their intention is to provide a basis for creating software systems. These software systems are created by using the middleware systems and extending their functionality.

Several CORBA implementations, e.g. TAO and Orbix, use the interceptor pattern to allow a flexible way of processing requests. Interceptors can be registered for different interception points, e g, request before or after marshalling.

Beside this implementation specific use, there is also a CORBA portable interceptor specification, which standardizes the use of interceptors

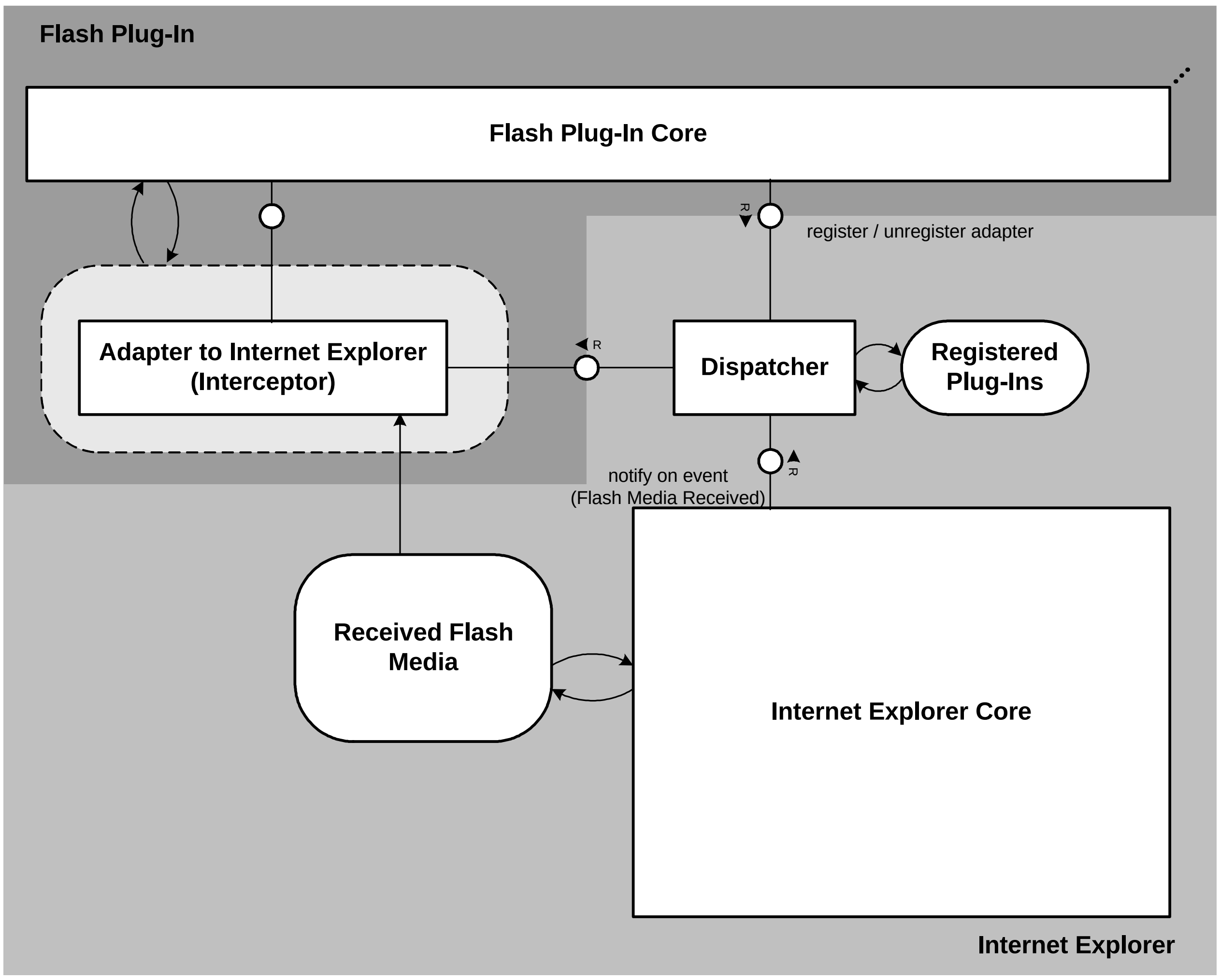


Figure 3: Compositional system structure showing web browser and plug-ins

for all implementations. This specification defines several interception points for whom interceptors can be registered.

The use of the interceptor pattern in COM is different from the one in CORBA. Interceptors in COM are only used to define a custom marshalling functionality for Objects. If an Interceptor is registered for that task, the custom marshalling is executed by this interceptor; if not the COM standard marshalling is applied.

Middleware systems are not the only applications of the interceptor pattern. Two other examples are described in more detail.

3.4.1. Web Browser Plug-In

Web Browsers, namely Internet Explorer or Netscape, implement the interceptor pattern to allow the integration of Plug-Ins. Plug-Ins are used to handle media types which the Browser itself cannot handle.

The following example depicts how Plug-Ins can register to different Web Browsers and how they are invoked. It does not go into detail on how Plug-Ins themselves can modify the state of a Web Browser.

Figure 3 shows how the browser and the Flash plug-in are connected. The Flash plug-in consists of the plug-in core, which essentially contains the Flash Player, and the browser adapter. This adapter corresponds to the interceptor of the pat-

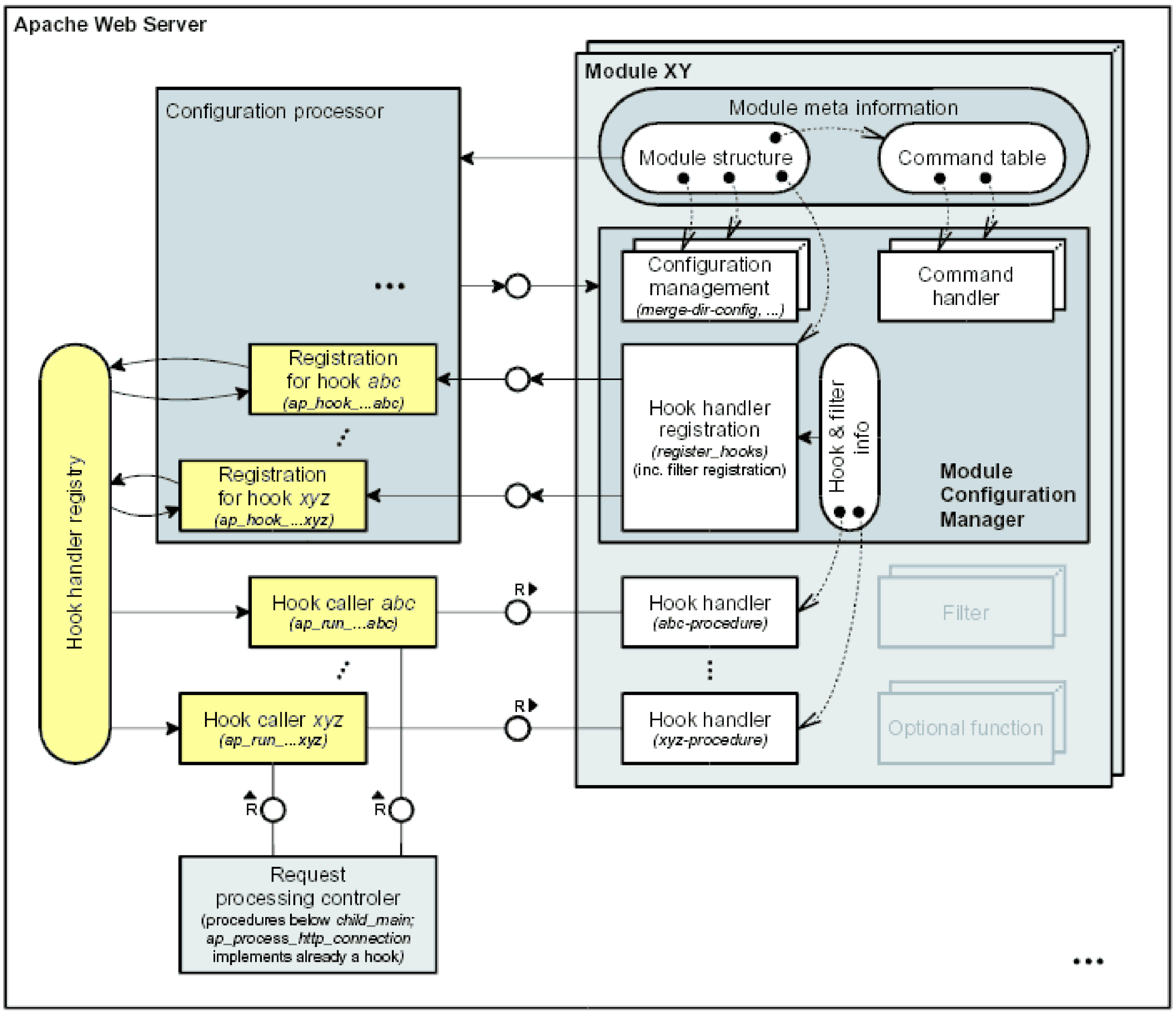

Apache Web Server
Configuration processor
Registration for hook abc (ap_hook_...abc)
Registration for hook xyz (ap_hook_...xyz)
Hook handler registry
Hook caller abc (ap_run_...abc)
Hook caller xyz (ap_run_...xyz)
R
Request processing controler (procedures below child_main; ap_process_http_connection implements already a hook)
Module XY
Module meta information
Module structure
Command table
Configuration management (merge-dir-config, ...)
Command handler
Hook handler registration (register_hooks) (inc. filter registration)
Hook & filter info
Module Configuration Manager
Hook handler (abc-procedure)
Hook handler (xyz-procedure)
Filter
Optional function

Figure 4: Apache Hook Handler registration and activation (taken from [GKKS03])

tern. By using adapters, one Flash core can be connected to different web browsers. It is invoked by the browser every time Flash media is received and transmits this media to the plug-in core which then displays it. Obviously this adapter has to be registered with the browser before this mechanism can work.

The browser has a component that is responsible for registering plug-in adapters. This component corresponds to the dispatcher of the interceptor pattern. This dispatcher also manages the list of registered plug-ins for different media types.

Whenever the browser receives media it cannot handle, an interception point occurs and the dispatcher is triggered. The received media is then passed to the dispatcher and from there to the adapter of the plug-in.

3.4.2. Apache Web Server

The Apache Web Server also implements the interceptor pattern. Apache 2.0 uses this to allow that modules can register handlers with the apache core.

Figure 4 depicts how this mechanism works in general. Any Module can register its handlers for a predefined hook. A hook is a specified “point of time” during the processing of the apache core, in which handlers can be called. These hooks represent exactly the interception points of the pattern.

The registration for handlers is done by the Configuration processor of Apache. This Configuration processor saves the information about all registered handlers in the Hook handler registry which also contains an order for these handlers. When an interception point (hook) occurs,

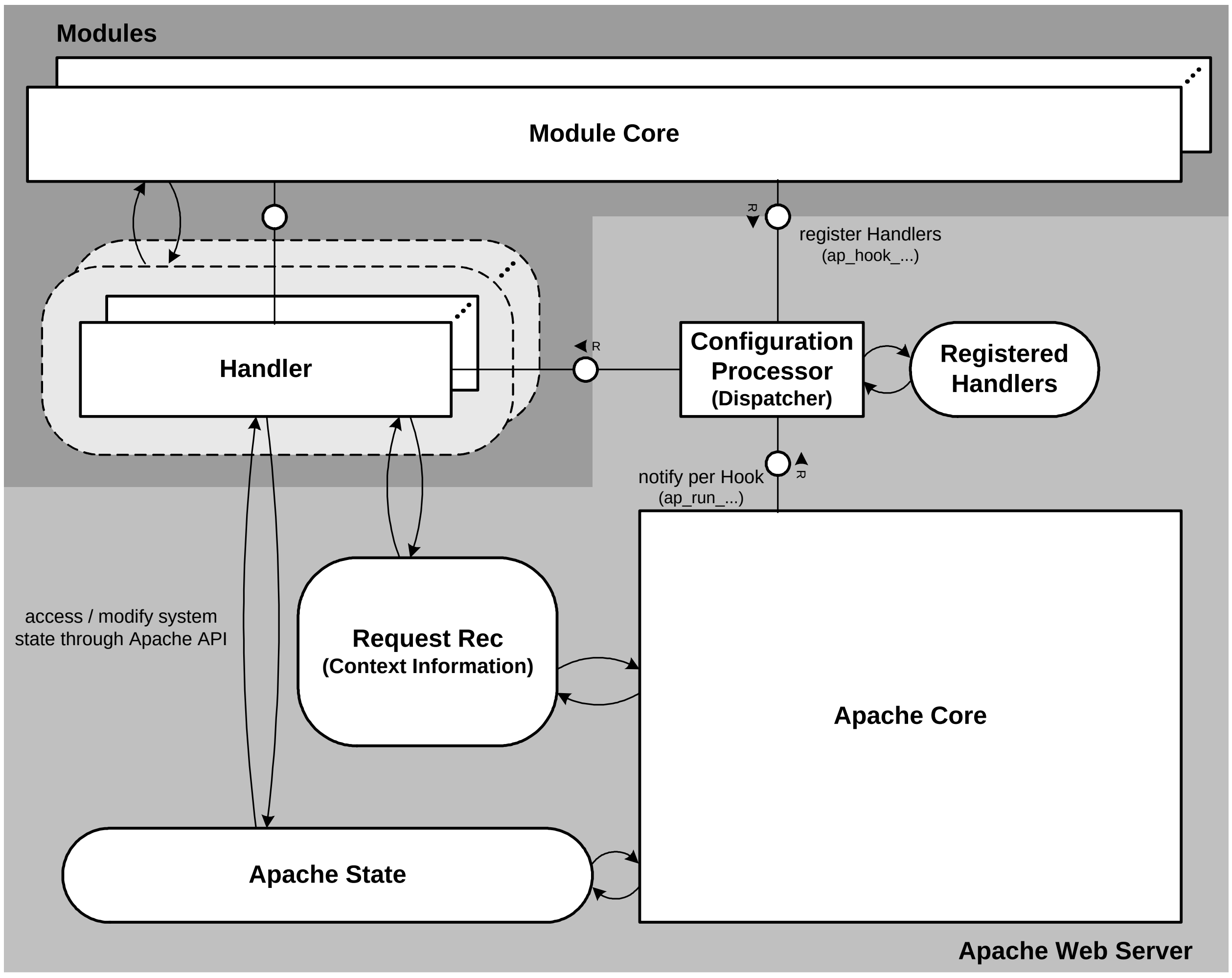


Figure 5: Compositional system structure showing Apache web serve and registered modules

Apache calls the handlers in the specified order, which can be unique for each hook.

The described mechanism is shown in Figure 5, so that it resembles the compositional diagrams discussed earlier.

A module is split into a Module Core and its Handlers. The Handlers are the interceptors and the configuration processor is the Dispatcher. The Module Core registers the Handlers via the Configuration processor which saves the registration information in the Hook handler registry. As the Hook handler registry contains ordering information for these handlers, the order for the currently registered hook has to be passed to the Configuration processor from the Module Core too. The Dispatcher is called every time a hook is reached during the processing of the apache core. The data that is passed on to the Handlers, the context, is in case of the Apache the Request Rec structure. This structure which contains all relevant information about the currently processed request can be modified by the handlers. Additionally the handlers have direct access to the Apache core via the Apache API. The Apache API is a set of methods that allow the modifying of the Apache State. This is why the access to the Core is shown here as a modification of the Apache state.

Apache even extends this mechanism in a way which is not shown in Figure 5. Handlers can define hooks themselves. This allows more flexibility as handlers can be called from within handlers when a hook is reached during the processing of a handler.

4. Discussion

4.1. Pros and Cons of the Interceptor Pattern

Applying the Interceptor architectural pattern offers a number of advantages:

- Interceptors decouple communication between sender and receiver of a request. Any candidate may fulfill the request depending on run-time conditions.
- Users can change a system’s functionality without changing its internal logic, possibly at runtime.
- The Interceptor pattern supports system monitoring and control through its context agent interface.

The openness of the concept also implies some disadvantages:

- System design gets more complicated. There is a tradeoff between extensibility and lean interface: introducing more kinds of interception may bloat interfaces but also makes the system more flexible.
- It is possible to introduce malicious or erroneous interceptors. The system may be more vulnerable.
- Unwanted interception cascades can occur when one event triggers a change in system state that in turn triggers other events.

4.2. Differences to Other Patterns

In contrast to its relative, the Chain of Responsibility pattern [GOF95], the Interceptor pattern allows more than one receiver to handle an event. Additionally it can offer applications modifying access to system behaviour.

Another pattern similar to the Interceptor is the Reactor pattern [POSA00]. In the Interceptor pattern the additional services, interceptors, do not have to be present as the control flow is independent of them. This is why the Intercpetor pattern describes a transparent extension of a system. The reactor pattern, however, is dependend on such extensions and is therefore not suitable for transparent event handling.

4.3. UML vs FMC Modeling

In our view using FMC to model a system containing the Interceptor pattern has proven successful. UML class diagrams capture mainly static code structures and do not support greater levels of abstraction well. With FMC it was possible to describe the runtime compositional structure of the system, which makes pattern principles more clear. Besides, the differentiation between “interceptors” and “concrete interceptors” like in [POSA00] became unnecessary. The figures presented here seem to be more instructive, universal, and also easier to understand than class or collaboration diagrams.

5. Conclusion

The Interceptor pattern can be used to solve problems concerning the extensibility of base systems by new services. It can therefore be found in many real-world systems, such as web browsers and servers, or middleware.

In our view using FMC for modeling has proven successful in capturing the essential pattern qualities in an implementation-independent way. A class diagram for the Apache web server would not even have been possible since it is not implemented with an object-oriented programming language.

6. References

- [POSA00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture Volume 2 – Patterns for Concurrent and Networked Objects.* Chichester: Wiley.
- [GOF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.
- [FMC] http://fmc.hpi.uni-potsdam.de
- [GKKS03] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, Oliver Schmidt (2003) *The Apache Modelling Project*. Forschungsprojekt am Hasso-Plattner-Institut Potsdam. http://apache.hpi.uni-potsdam.de/document/the_apache_modelling_project.pdf

Architecture Pattern
The Reactor

Nikolai Cieslak, Dennis Eder

Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany

{nikolai.cieslak, dennis.eder}@hpi.uni-potsdam.de

Abstract

This paper presents the Reactor Pattern as described by Buschmann et al [POSA2000] derived from the ACE Project [ACE]. It gives a brief overview of the essential components. We analyze alternative modeling approaches and the most common implementation examples. Finally, the Reactor is compared to other patterns.

Keywords: Pattern, Architecture, Event Handling, Reactor, Dispatcher, Demultiplex

Introduction

Event-driven applications in a distributed system must be prepared to handle multiple service requests simultaneously.

Before executing specific services sequentially, an event-driven application must demultiplex and dispatch the concurrently-arriving indication events to the corresponding processor components.

The Reactor Pattern is a common measure to handle occurring events and dispatch actions to other components, which have been registered in order to perform those tasks.

1. The Pattern

1.1. Principle

In order to handle occurring events and dispatch actions to other components, the Reactor keeps a list of handles[1]. The components register handles and their respective event handlers with this list. The concrete handler will be called when its handle gets into a signaled state. Until this call, the event handler stays inactive and does not consume any processor time.

The Reactor performs an event loop, in which it waits for any of the registered handles using a blocking call or polling the list. If one or more of these handles get signaled, the Reactor will parse the list and call for each signaled entry its respective event handler.

1.2. UML- Modeling

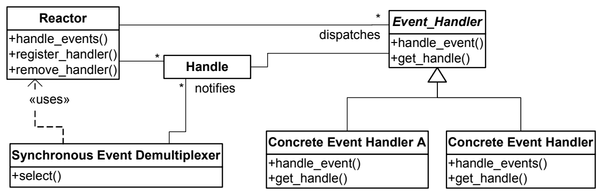


figure 1 – UML Model of the reactor pattern [taken from POSA2000]

This UML Model shows the structure of the Reactor Pattern like it would be implemented in an object oriented language.

The model shows that an event handler owns a handle, which can be registered with the Reactor and that the Reactor dispatches the event handler and waits for events using an event multiplexing mechanism.

[1] Handles are operating system resources like files, communication ports etc. It is possible to wait for handles using a blocking call, so that task switches can be reduced. In that case the operating system will wake up the task when an event occurs.

1.3. FCM Modeling

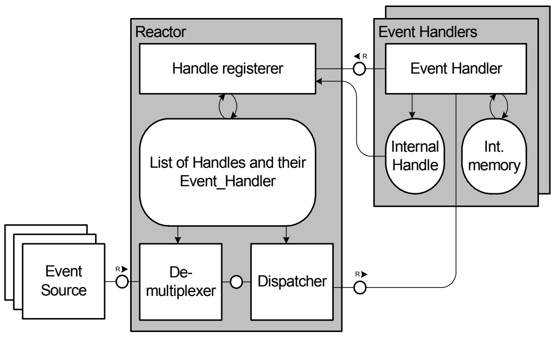


figure 2 – FMC block diagram of the reactor pattern

The FMC model illustrates that an event handler consists of a handle, which is registered with the Reactor and stored in its "list of handles", the handler, which is called by the dispatching mechanism of the Reactor, and internal memory.

It also displays that the Reactor is listening to occurring events on event sources using a demultiplexing mechanism.

1.4. Where can we find the pattern?

The use of the pattern is not limited to object oriented languages as the use of an UML diagram could imply. The Reactor Pattern is a more general pattern and can also be found in low-level programming like operating systems. The Reactor Pattern is used for implementing software interrupts (signals) in UNIX. You also find the Reactor Pattern in the hardware interrupt handling of "Microsoft Windows 2000" that uses interrupt service routines, which can register themselves and which will be called if the interrupt occurs.

Another common use of the pattern is the implementation of servers. The pattern can be used to have one listening process, which waits for occurring events or jobs and dispatches them to other routines.

Examples for this implementation can be found in the Apache web server or the INETD.

2. The INETD

The INETD (InterNET superDaemon) is a simple daemon used in UNIX to implement a set of small network services.

2.1. Principle

The INETD listens to a set of communication ports (operating system handles) and dispatches external programs or internal routines depending on which port a connection request arrives. The corresponding connection handle is passed to the external program, so that the other program can communicate with the opposite side.

In order to wait for connection requests, you only need to have one program (the INETD) running. After a connection has been established, also external programs will be running until the connection is closed again.

2.2. Usage of Reactor Pattern in INETD

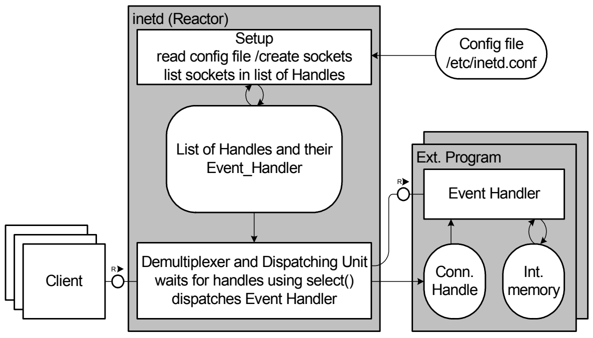

inetd (Reactor)
Setup
read config file /create sockets
list sockets in list of Handles
Config file /etc/inetd.conf
List of Handles and their Event_Handler
Ext. Program
Event Handler
Client
Demultiplexer and Dispatching Unit
waits for handles using select()
dispatches Event Handler
Conn. Handle
Int. memory
R

figure 3 - FMC block diagram of the INETD

The FMC block diagram in figure 3 shows a structure differing slightly from the pattern:

The event handlers do not register themselves with the Reactor but a configuration file is read and the handles are created by the setup routine. When the connection request arrives the INETD establishes the connection and starts a new process by using fork() and exec(). This process has access to the connection and executes the associated program.

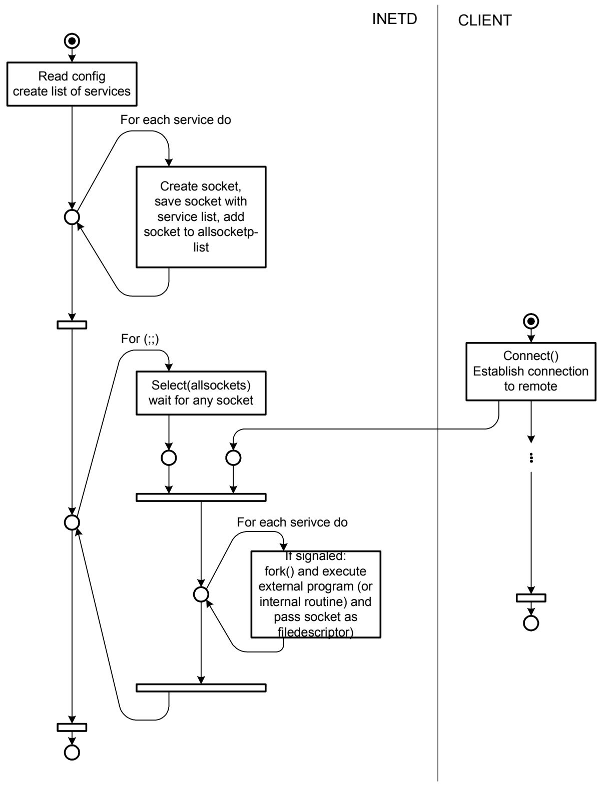

INETD
CLIENT
Read config create list of services
For each service do
Create socket, save socket with service list, add socket to allsocketp-list
For (;;)
Select(allsockets) wait for any socket
Connect() Establish connection to remote
For each serivce do
If signaled: fork() and execute external program (or internal routine) and pass socket as filedescriptor)

figure 4 - FMC sequence chart of the INETD

When the INETD starts a configuration file ("/etc/inetd.conf") is read. For each entry of the configuration, the appropriate sockets are created and the handles are stored in a set of handles.

The INETD waits using *select()* for any handle of the list of handles to get into a signaled state. If any handle has reached a signaled state the INETD continues returning from the blocking call of *select()*. It checks each handle of the list whether it is signaled or not. If it is in a signaled state the INETD accepts the connection and starts the corresponding ... () using fork() and exec()[2].

After all handles have been checked against signaled states, the INETD will restart its event loop and waits again using *select()* for new occurring events.

3. Pattern Variants

There are many possible variations of the pattern. This paper presents two very handy and frequently used ones.

[2] INETD links the socket representing the connection to the file descriptors 0,1 and 2 (STDIN/OUT/ERR) of the new process. The event handler therefore has to use these file descriptors to read and write to the client.

3.1. Concurrent event handlers

Normally, the pattern incorporates a single-threaded reactive dispatching design. That means that handlers use the thread of control of the reactor.

In this variant, event handlers can run in their own thread of control (see INETD). This allows the reactor to demultiplex and dispatch new indication events concurrently with the processing of previously dispatched event handlers.

In FMC, this can be modeled as multiple event handler actors coordinated via multiple request channels by the Dispatcher. For simplicity, we merge them into a single unit, as shown in figure 5.

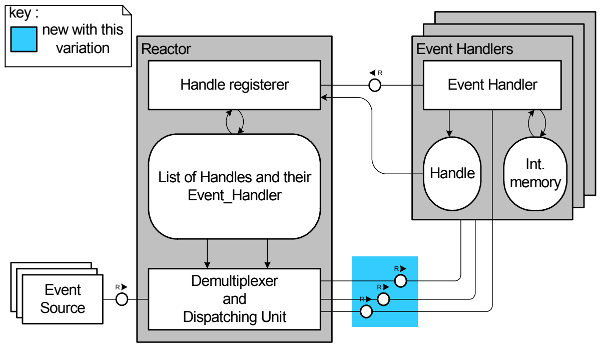


figure 5 - FMC block diagram of the Reactor Pattern with concurrent event handlers

Patterns, which are recommended for implementing concurrent concrete event handlers, are

- Active Object
- Leader / Followers
- Half-Sync / Half-Async

Details follow in the Composition Section.

3.2. Concurrent synchronous event demultiplexer

In its standard version, the single synchronous event demultiplexer is called serially by a reactor in a single thread of control.

In this variant several synchronous event demultiplexers are called concurrently on the same handle set by multiple threads. It obviously correlates with the same capability of the Win32 WaitForMultipleObjects() function. The corresponding FMC model can be derived from figure 5 by adding multiple demux/dispatch units. This is shown in figure 6.

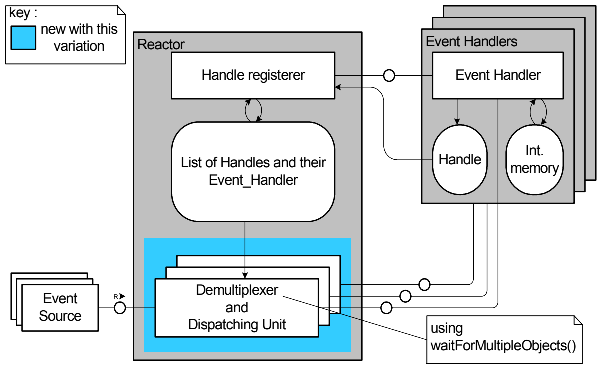


figure 6 - FMC block diagram of the Reactor Pattern with concurrent synchronous event demultiplexers

The main advantage of this variation is, that it improves application throughput, by allowing multiple threads to simultaneously demultiplex and dispatch events to their event handlers.

On the other hand, its implementation can become much more complex and less portable, such that the Dispatcher might have to perform reference counts or the Reactor might have to queue calls to the Reactor's procedures for registering and removing event handlers (Command Pattern[GoF95]) to defer changes until there is no thread dispatching an event handler.

4. Pattern Composition in Apache

4.1. Acceptor/Connector

The reactor in combination with the Acceptor/Connector pattern [POSA2000] is common for designing server applications that use TCP, such as INETD. The Acceptor/Connector pattern basically correlates with the Gatekeeper pattern [PGM]. This paper focuses on latter pattern, since it has already been modeled in FMC. The demultiplexer and the event handlers of the Reactor can be mapped to components of a fine-grained Gatekeeper[PGM] model as shown in figure 7.

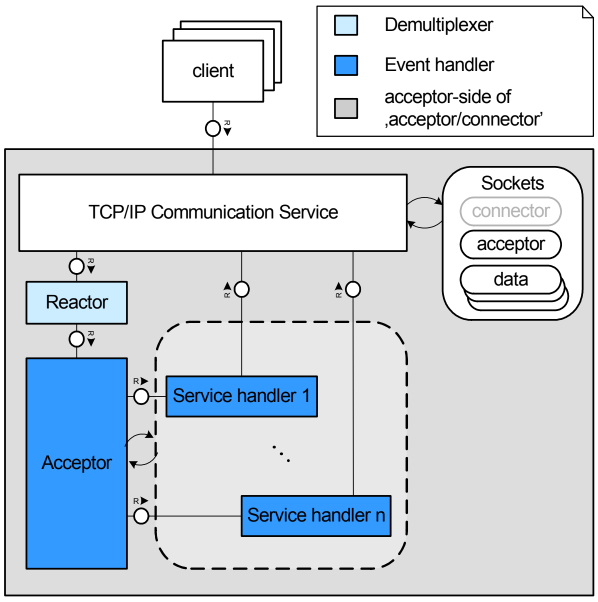


figure 7 - FMC block diagram of Acceptor / Connector with Reactor

Apache ships with a Worker MPM. In order to identify pattern components in this MPM, it is useful to think of a two-level Reactor. Demultiplexing, dispatching and handling is done twice for each request namely at listener and worker level as shown in figure 8.

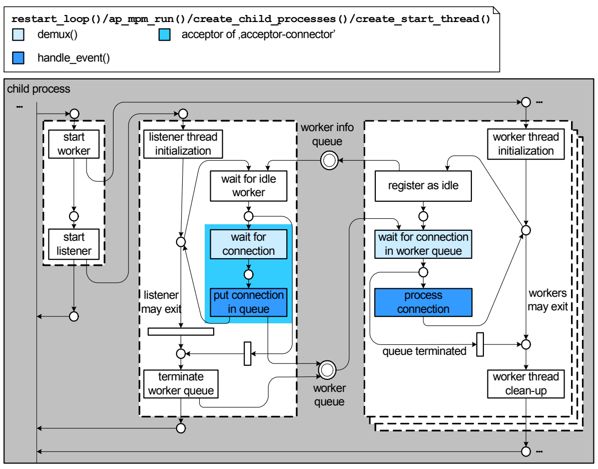


figure 8 - Behavior of Apache's Worker MPM – dynamics of an Acceptor / Connector ?

Unfortunately, the design as outlined here has some performance drawbacks on massive traffic. The reason is probably the queue construction. All idle threads block on a single shared 'condition variable' (pthread) until a new job becomes available. When triggered, an unspecified notify() of any blocked thread wastes resources.

Bushmann et al [POSA2000] propose the following abstract solution: Each thread has its own condition variable so that a single, suitable thread can deterministically be notified.
Two example implementations of this derivate can be found in the Apache httpd : the Leader MPM and the Threadpool MPM.
Details on both MPMs follow next.

4.2. Leader/Follower

The Leader/Follower pattern provides an efficient concurrency model. Its structure contains a pool of threads to share a set of event sources by taking turns in demultiplexing events that arrive on these event sources and synchronously dispatching the events to application services, which process them. More details can be found in [POSA2000].

There is a problem with its standard implementation. Implementing a coarse-grained Leader/Follower design burdens high complexities: Programming involves mechanisms to demultiplex handle sets. These in turn, involve native operating system calls. Even worse, all that must be done in a highly concurrent context.

Applying higher-level patterns to compose a fine-grained Leader/Follower design makes it easier to decouple the I/O and demultiplexing aspects of a system from its concurrency model, thereby reducing code duplication and maintenance effort. figure 9 illustrates the actor-role-dynamics specified by the Leader / Follower pattern in conjunction with the reactor's components, as implemented in the Apache Leader MPM.

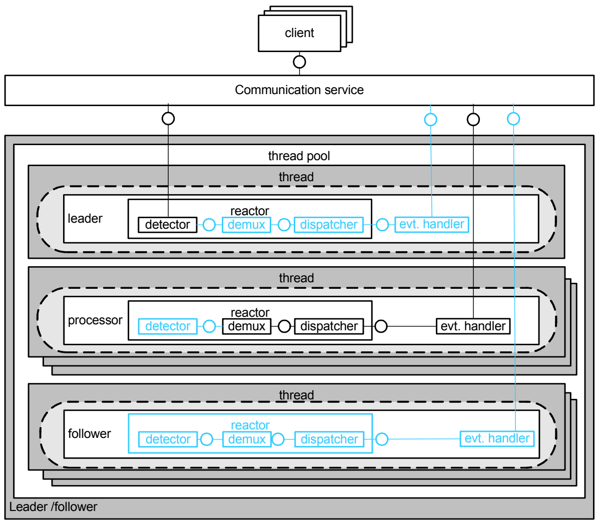


figure 9 FMC block diagram of Leader / Follower Pattern

The thread pool contains n threads with variable structures. Roles specify the concrete structure inside of the threads' structure variance. Every thread can have one of the three roles at a time. Only one thread at a time can have the leader role, and thus wait ('detect') for an event to occur on a set of event sources. In the figure temporarily inactive components are colored blue.

4.3. Half-Sync/Half-Reactive

The Half-sync / Half-Async pattern [POSA2000] decomposes services of a modeled system into two layers (synchronous and asynchronous) and adds a queuing layer between them to mediate intercommunication of the two layers.

Half-Sync / Half-Reactive [POSA2000] is an implementation of the Half-sync / Half-Async pattern. It combines the reactor's event demultiplexing components with the thread pool variant of the Active Object pattern [POSA2000]. Latter pattern decouples method execution from method invocation.

Again, an example implementation can be found in an Apache MPM - namely the Worker MPM. It strongly correlates with the Worker / Listener Pattern [PGM]. Latter has already been modeled in FMC. Therefore, we use it as the underlying structure in figure 10.

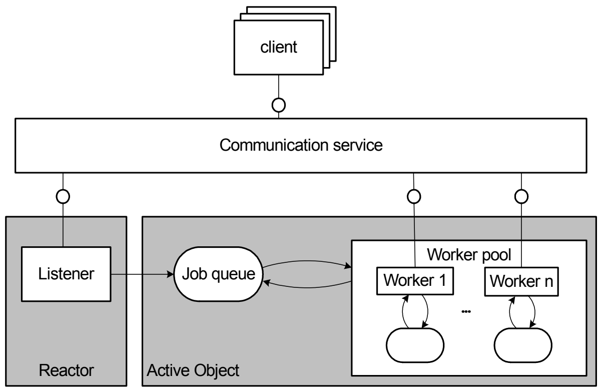


figure 10 - FMC block diagram of Half-Sync / Half-Reactive Pattern

The architecture's partition into Reactor and Active Object, as proposed by the Buschmann et al [POSA2000], is modeled as two grey areas.

5. Separation from other patterns

5.1. Proactor

Buschmann et al [POSA2000] consider the reactor pattern as a synchronous variant of the asynchronous Proactor pattern. Like the Reactor, the Proactor supports the demultiplexing and dispatching of multiple event handlers.

In the Reactor, these events are triggered when indication events signal that it is possible to initiate an operation synchronously without blocking.

In contrast, in the Proactor, events are triggered by the *completion of asynchronous operations*.

5.2. Interceptor

In general, the Reactor as well as the Interceptor pattern allows services to be transparently added and triggered automatically when certain events occur.

However, the Reactor is merely thought of as an event-handling pattern, which tackles the task of demultiplexing concurrent events from one or more clients. Consequently, a Reactor's dispatcher forwards each event to exactly one event handler.

In contrast, the Interceptor's core capability is to access and configure 'out-of-band' services in a framework. These services can then intercept at any predefined point of the framework and can cut across multiple layers in the architecture. A key aspect is the services' ability to control a concrete framework's subsequent behavior, when a specific event occurs. In particular, services can manipulate event contexts, a component type not found in the Reactor. A dispatcher in the Interceptor pattern usually forwards events to *all* concrete interceptors that have registered for it.

In practice, the Reactor pattern focuses on handling system-specific events occurring in the lower layers of a communication framework, while the Interceptor pattern helps to intercept application-specific events at multiple layers between the framework and the application.

6. Discussion

6.1. What are the main aspects of the Reactor?

In our opinion, the reactor is a general pattern for an event handling demultiplexing process. We do not share the presentation of Buschmann et al [POSA2000], which suggests that it is an object-oriented pattern.

We have found usages of the pattern throughout all kinds of application, including internet servers (INETD, Apache) and operating systems (Windows 2000 and Unix). Although the implementation differs slightly from Buschmann et all [POSA2000] we still consider it as the Reactor Pattern.

6.1.1. Inetd

The INETD uses a slightly modified variant of the pattern.

The handlers do not register themselves with the Reactor, but the list of handles and their handlers is built from a configuration file. But for all that, it is still about handling events using a demultiplexing and dispatching mechanism and for this reason still the Reactor Pattern in our opinion.

6.2. Nomenclature of Pattern and Components

The pattern and a component of the pattern carry the name Reactor. This could be quite confusing to differentiate, which components are considered as part of the pattern.

In our opinion, the Reactor is the central part of the pattern and all other components are the environment of the pattern. Thus, the environment could be modified slightly without violating the pattern itself.

In contrast to Buschmann et al [POSA2000] we model the dispatcher as a subcomponent of the Reactor component as we do with the Demultiplexer. This contrast derives from Buschmann's object-oriented view, and consequently from his implementation-oriented considerations about class granularity whereas we apply the somewhat more abstract FMC.

6.3. Detector

Arguably, the demultiplexer in most implementations will merely consist of a simple system call. Still, it is explicitly modeled as a means of conceptual modularization. This also allows easy substitution by a high-level demultiplexer.

6.3.1. Leader Follower

In that sense, the Leader/Follower pattern points to a similar problem. The pattern denotes 'detection' as an integral part of the entire request processing, just before the demultiplexing phase. Thus, it has been included in figure 9 as a Reactor's component. It seems to be the only way to correctly model the concurrent access of a Leader and n Processors to the Reactor Singleton. This is important to localize thread-safety issues.

But in most cases, such a fine-grained view of the Reactor is unnecessary, also in that 'detection' is already implemented by communication services or others. However, Buschmann et al [POSA2000] do not mention this composition problem at all.

7. Conclusion

The Reactor pattern is used in many different ways. It is not limited to the object-oriented world. As shown, it is used in procedural programming, too.

The Reactor's serialized, single-threaded event loop can simplify the coordination of otherwise independent event handling services. Variants on the other hand, may bust the event loop's simplicity, but are tolerable to the extend presented above.

In trivial implementations, the Pattern is part of the operational system for implementing software interrupts. In that case, architects and programmers alike might still gain from the Reactor's conceptual encapsulation of native event demultiplexing mechanisms, such as Select(), WaitForMultipleObjects() and WaitForSingleObject().

From a higher-level point of view, it is particularly worth the extra structuring effort, when effectively combined with other patterns to support concurrent handling of events.

References

[AMP] The Apache Modeling Project, http://apache.hpi.uni-potsdam.de

[ACE] The ACE Project http:// www.cs.wustl.edu/~schmidt/ACE.html

[POSA2000] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-Oriented Software Architecture. Wiley, 2000

[PGM] B. Gröne & A. Knöpfle, Pattern-Grafische Muster, Beschreibungsmuster & Systemmuster, Hasso-Plattner-Institute, Potsdam University 2000

[Douglas1] Douglas C. Schmidt, Reactor – An Object Behavioral Pattern for Demultiplexing and Dispatiching Handles for Synchronous Events, Washington University, St. Louis, http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf

[Douglas2] Douglas C. Schmidt, Applying Patterns and Framworks to Develop Object-Oriented Communication Software, Washington University, St. Louis

Half-Sync/Half-Async Concurrency Pattern

Robert Mitschke, Harald Schubert
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{Robert.Mitschke, Harald.Schubert}@hpi.uni-potsdam.de

Abstract

Concurrency issues are among the hardest tasks to be dealt with when designing a software system. While a high concurrency degree may improve the system performance it also bears the risk of further difficulties due to the increased system complexity.

Connected to concurrency issues is the question of whether services provided by the system should be processed synchronously or asynchronously. At large, the freedom to actually choose between the two alternatives depending on the situation seems most appropriate.

With the Half-Sync/Half-Async concurrency pattern, we present a model of an architectural pattern that allows services to be processed partly synchronously and partly asynchronously.

Keywords: Patterns, Architecture, Concurrency, Synchronous, Asynchronous, Modeling

1 Introduction

“Concurrent systems often contain a mixture of asynchronous and synchronous service processing” [POSA2000]. Taking this citation into consideration, particular large-scale and complex concurrent systems perform both synchronous and asynchronous processing for different reasons:

- Synchronous processing is generally used to simplify programming.
- Asynchronous processing is often required due to performance reasons.

Since the terms *synchronous* and *asynchronous* are of substantial relevance for the closer examination of the Half-Sync / Half-Async pattern, a formal definition of these terms is given in order to avoid misunderstandings and to clarify their meaning.

In the context of both forms of processing, the Half-Sync / Half-Async pattern introduces a possibility to combine the need for asynchronous processing and the benefits of synchronous processing and to enable communication between those two types of services. Therefore, three layers are established, a synchronous layer containing the synchronously processing services, a corresponding asynchronous layer and a mediating queuing layer.

In the following chapters, the architectural components used to realize the pattern are modeled in detail. Examples are given to improve the understanding of the pattern. Finally, a discussion about the method used to model the pattern and a comparison to related patterns conclude the paper.

2 Synchronous and asynchronous

The terms *synchronous* and *asynchronous* are used in various fields of applications such as natural sciences, sports and even everyday life. While our common understanding of *synchronous* is to be seen as more or less equivalent to that of the word *simultaneous*, we will see that this definition is not accurate enough for our purpose.

Both terms refer to the way executing entities perform their actions with respect to each other. The key criterion to determine whether these actions are to be seen as synchronous or asynchronous lies in their degree of temporal coordination.

For a formal definition of the terms we use an approach based on modified petri nets presented in [FMC].

2.1 Synchronous

The expression *synchronous* means that the actions are performed in a temporally coordinated manner. Imagine a group of swimmers performing synchronous-swim, moving simultaneously. Even if the swimmers swam in a temporally shifted manner one would still say that they do swim synchronously. Transferred to the context of programming, functional system components correspond to the executing entities mentioned above. Components, which from a code view might be represented as two modules, with a module being a set of procedures and variables, communicate with each other by means of procedure calls. These procedure calls though will usually result in the execution of actions being temporally shifted and not simultaneous, reflecting the fact that a processor can only execute one instruction at a time. Interpreting the petri net in figure 1 as a procedure call after instruction a1, transferring control from component A to B, gives us a good example for this way of looking at things. After the transfer of control, component A is inactive until component B has finished its processing and only then continues with its execution of instruction a2. This behavior is synchronous due to the temporal coordination. Thus, it seems appropriate to generalize the definition of synchronous and to consider the simultaneous execution of actions as a special case of synchronous execution.

Taking this definition for granted we now have to find a way to clearly define what temporally coordinated exactly means. As already mentioned, we use an approach based on modified petri nets.

As described before, figure 1 shows two entities A and B which carry out two actions a1, a2 and b1, b2, respectively. Using two synchronization points within the petri net defines exactly one order of execution for these actions. There is only one possible event series: (a1, b1, b2, a2).

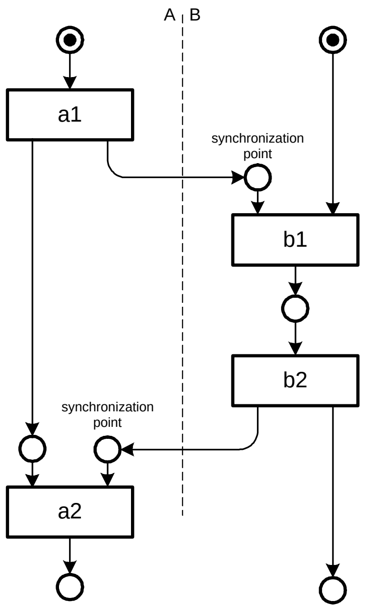


Figure 1 Synchronous Processing

As a formal characteristic, actions are performed synchronously if the corresponding petri net allows only one order of execution.

Note that a trivial case of synchronous execution is given if the number of considered entities equals one: An entity is always synchronous to itself.

2.2 Asynchronous

Analogously, the word *asynchronous* means that the actions are not temporally coordinated. Like in a badly organized teamwork where teams do not cooperate properly, leading to poor performance and misunderstandings, the actions are uncorrelated.

As to the formal definition, asynchrony is given if the petri net describing the possible event series for the considered actions does not lead to a single order of execution. For example, figure 2

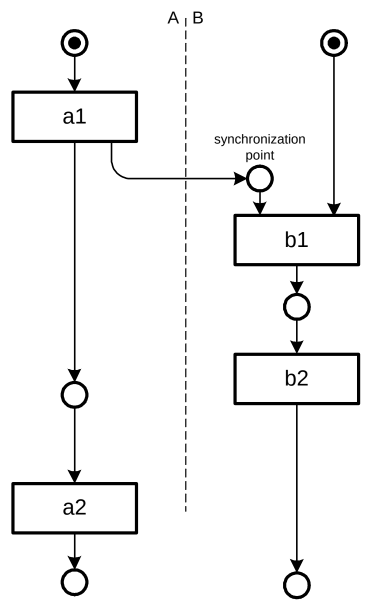


Figure 2 Asynchronous Processing

lacks one synchronization point, thus resulting in three possible event series: (a1, b1, b2, a2), (a1, b1, a2, b2) and (a1, a2, b1, b2). An equivalent characteristic is whether at any given point in time while processing the petri net there are concurrently possible transitions.

As with synchronous processing there is also a trivial case in case of asynchronous processing: two or more entities that are completely decoupled, which implies that there are no synchronization points, perform their actions asynchronously.

Having defined the terms synchronous and asynchronous on the basis of temporal coordination, we now continue with the presentation of the Half-Sync/Half-Async pattern.

3 Pattern Core Components

Considering highly complex concurrent systems, often a combination of synchronously and asynchronously processing services is provided in order to realize the system functionality. Especially higher-level system components make use of synchronous processing, thus avoiding the complexities of asynchrony and simplifying programming. In contrast to higher-level components, lower-level system components, often close to hardware-related topics, are mainly implemented asynchronously, either for performance reasons or, as in the case of hardware issues, due to time requirements like interrupt servicing.

These reflections lead to an intended system that allows synchronous and asynchronous services to coexist and to communicate with each other (figure 3).

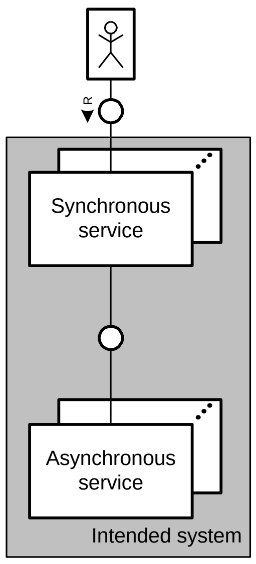


Figure 3 Intended System

In order to realize this intended system, an additional layer is introduced. Its task is to serve as an intermediary between the synchronous and asynchronous services. It provides a queuing mechanism that decouples the request of one service from the servicing of another service (figure 4).

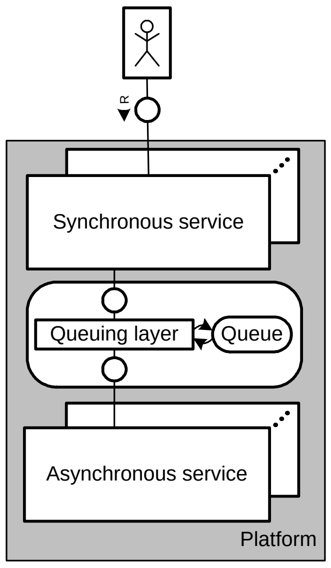


Figure 4 Platform

At this point communication is reduced to basic message passing: two components are considered to be able to communicate if they can send messages to one another. The following diagrams show the flow of control in the message passing procedure for both directions: asynchronous to synchronous (figure 5) and vice versa (figure 6).

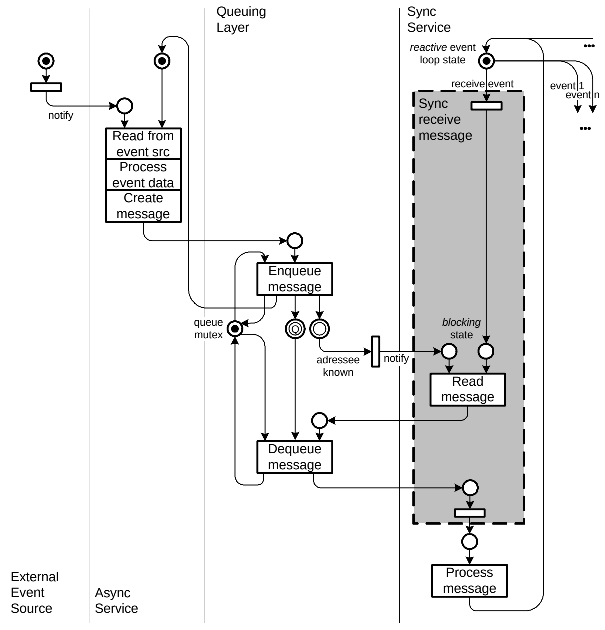


Figure 5 Async to Sync Communication

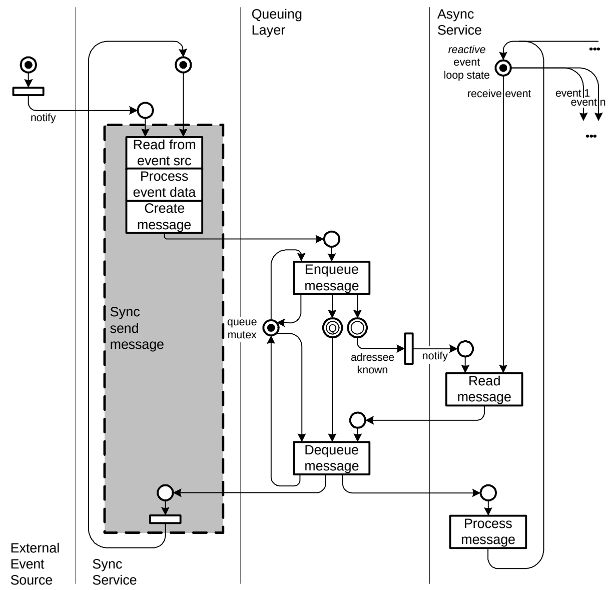


Figure 6 Sync to Async Communication

At the beginning of our observation, there is an event which notifies a service, such as an event of an external event source. Associated to this event is a message which is read by the notified service. The service processes the message and will finally create and enqueue a message into the queuing layer.

Note that whether a service of an adjacent layer is considered synchronous or asynchronous is to be seen in the context of the queuing layer: If a service blocks on the call of the queue (Receive in figure 5, send in figure 6), storing or retrieving a new message, he is temporally coordinated with the queue. If he does not block (Send in figure 5, receive in figure 6), he works asynchronously.

Thus, depending on whether the service is synchronous or asynchronous, he will or will not wait until the message has been read by the addressee or, if no specific addressee is defined, by a random service. On the other side of the queue, depending on the service's type, the service will or will not block on the procedure of receiving the message. On both sides of the queue, the asynchronous variant allows more responsiveness since the service does not block on an event to happen.

Note that this model of communication has been simplified to a degree suitable for the purpose of the examination. The aspect that synchronous message passing (e.g. network messages) often implies some sort of response mechanism including information on the result of the message handling, has been omitted.

In this chapter we outlined a queuing mechanism used to provide an environment with which synchronously and asynchronously processing services can intercommunicate. Next, we present examples to illustrate the pattern.

4 Examples

4.1 A Fast Food Restaurant

Thinking about fast food restaurants gives us a good example of how the Half-Sync/Half-Async pattern is used in everyday life.

As it can be seen in figure 7, there are two types of agents within the restaurant: waitresses and cooks. Further, there are two queues: a burger storage place and an order storage place.

The waitress works synchronously, waiting for a customer to appear and to place his order. Obtaining an order, the waitress looks at the burger storage place and collects the products. In case a burger is currently not available, the waitress has the option to forward the order to one of the cooks, putting a message into the order storage place and then waiting for the product to arrive in the burger storage place. Finally, the waitress receives the money for the meal and only then is ready to serve a new customer.

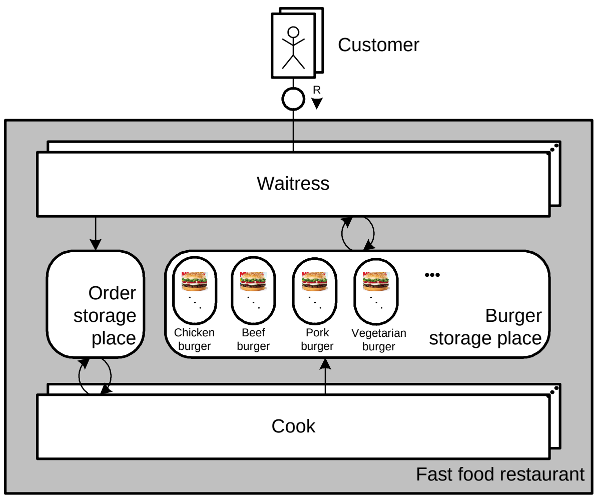


Figure 7 A Fast Food Restaurant

Behind the burger storage place, the cook works asynchronously. For the most part preparing several burgers at a time and turning over the meatballs as soon as they are done, he does not wait for a specific order to arrive, which would be synchronous behavior. Instead he always keeps a specific quantity of each product ready to be sold. Only periodically he will check the order storage place to prioritize his work.

4.2 Sockets

A well-known example for an implementation of the Half-Sync/Half-Async pattern is the socket mechanism used in contemporary operating systems.

Almost all operating systems which provide multiprocessing or multithreading also offer a way to react synchronously to asynchronous events. Accessing a hardware device or reacting to any kind of unpredictable input from outside the system requires the synchronization of mainly synchronously processing applications and asynchronous system services. Events related to networking issues belong to this category of asynchronous events.

The use of the Half-Sync/Half-Async pattern for the implementation of the socket mechanism is based on employing the operating system kernel to put threads to sleep and to wake them based on the occurrence of events.

In order to send information via the network, the synchronous application calls the kernel to request the asynchronous network service for service. The kernel then takes the request and immediately puts the synchronous application thread to sleep. It will also queue the request to the asynchronous network service. Only after the asynchronous service finished its task and after the result of the request is available, the synchronous application thread is woken and can gather the result continuing to compute its operation synchronously.

In case the synchronous application wants to receive information it will analogously address a corresponding request to the kernel. The kernel will then put the application thread to sleep, queue the request to the asynchronous network service and only after it received the information from the asynchronous network service it will wake the synchronous application thread and provide the result.

In both cases the application will remain synchronous from its point of view. Pausing and resuming the thread is transparent for the application thread itself.

The asynchronous layer is usually implemented as part of the operating system and uses asynchrony techniques that are supported by the hardware platform. These usually include software and hardware interrupts that allow notification and event handling for asynchronous events.

4.3 WinNT IOCP

The Windows NT operating system mechanism *I/O Completion Port* implements a variant of the Half-Sync/Half-Async pattern: the Half-Async/Half-Async pattern.

The IOCP does not support the communication between synchronous and asynchronous services but rather propagates asynchrony to higher level asynchronous services.

Basically, the IOCP is a combination of a job queue and an idle thread queue. It is supplied by the operating system and needs the kernel to provide its functionality.

The IOCP allows multiple threads to request the IOCP to take care of an asynchronous I/O operation. However, the call to the IOCP is non-blocking. After asking the IOCP, the thread can continue with other operations. Therefore, he is considered asynchronous.

On the other hand, whenever a thread is ready to process the result of any of the requests left with the IOCP, it can offer its service to the IOCP. If no I/O completion is available, the requesting thread is blocked meanwhile.

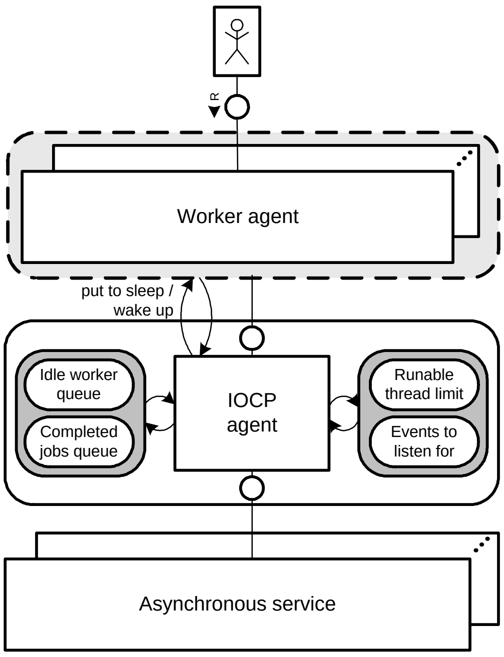


Figure 8 Structure of the IO Completion Port

The advantage of this approach is the number of threads processing I/O operations may be decoupled from the number of I/O operations.

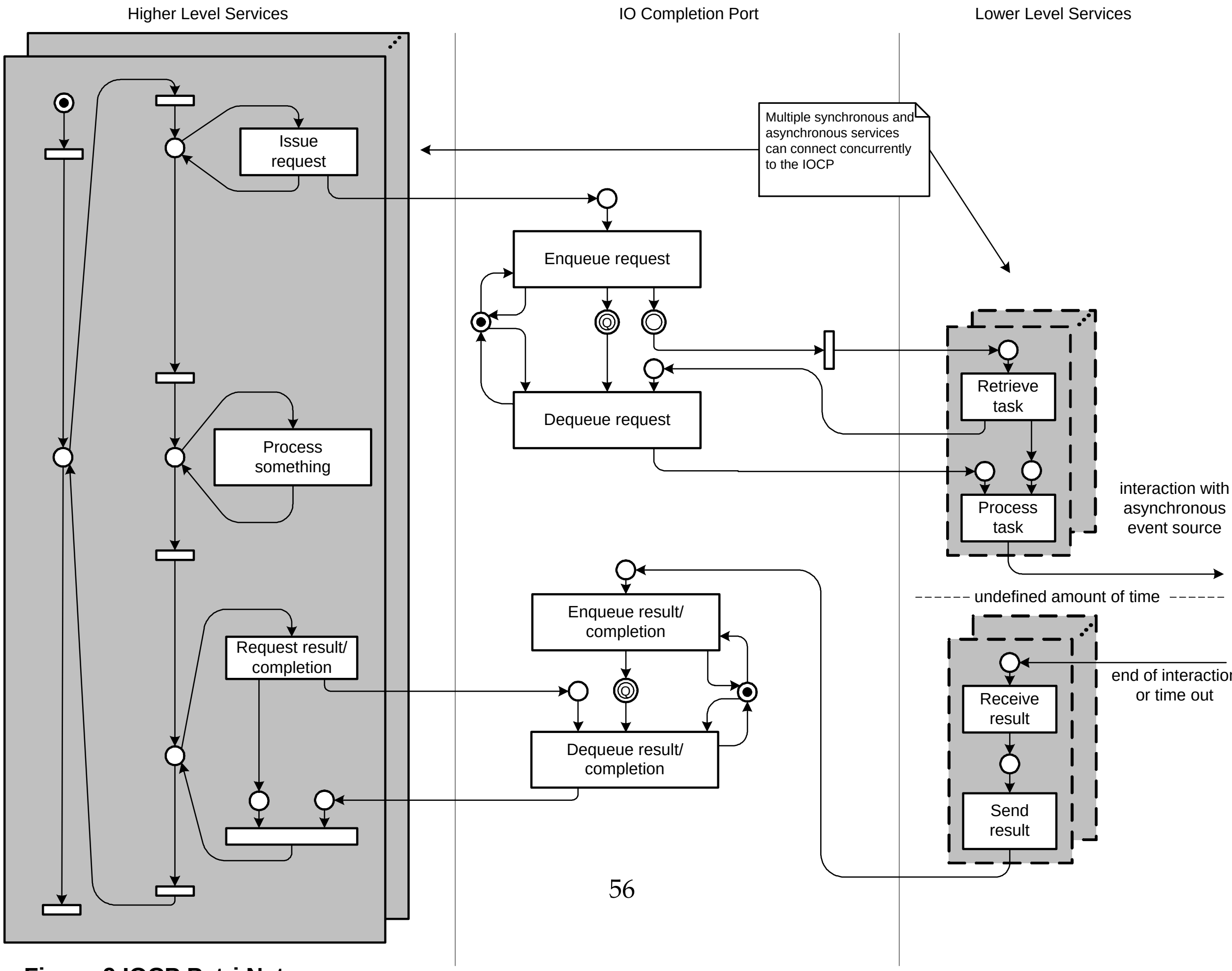
The IOCP additionally adds support for restricting the quantity of active worker threads. It checks the status of all worker threads currently registered with the IOCP. In case a certain amount of threads is already active, threads returning from servicing a request are put to sleep.

This is done to minimize paging efforts and to minimize the need for context changes, since scheduling activity is minimized if only a few threads are active.

A possible application of the IOCP is a web server. A web server which mainly serves a potentially large amount of requesters has to spawn a lot of threads if many requests arrive. Additionally, handling one request is a short procedure and most tasks that incorporate a handling of the request are again blocking calls.

Using IOCP a web-server needs only a small number of threads to serve a potentially unlimited number of clients. Each time a request arrives and the limiting value of concurrently processing workers is not reached, a worker thread will receive the request and start handling it. It will soon encounter a call to an asynchronous service, like fetching a file from the hard disk, and can reregister that call with the IOCP and serve another client request.

The major challenge which the IOCP approach includes is that the state of a single request handling activity is no longer implicitly stored in



Higher Level Services
IO Completion Port
Lower Level Services
Issue request
Process something
Request result/ completion
Multiple synchronous and asynchronous services can connect concurrently to the IOCP
Enqueue request
Dequeue request
Retrieve task
Process task
interaction with asynchronous event source
undefined amount of time
Enqueue result/ completion
Dequeue result/ completion
Receive result
Send result
end of interaction or time out

Figure 9 IOCP Petri Net

the threads sequence of execution. Therefore the higher level services need to store the state information in a proprietary way. To manage this, the IOCP concept includes means to match a completion with a specific event source.

This context is shown in figure 9. Lower-level as well as higher-level services work asynchronously since neither of them blocks on a call of the IOCP. The fact that the thread issuing a request does not necessarily have to be the thread which handles the corresponding completion afterwards is shown by means of multiple instances of synchronous and asynchronous services.

5 Modeling the Pattern

When modeling the Half-Sync/Half-Async concurrency pattern, the main task consists of modeling dynamic structures since the static structure of this pattern is reduced to an ordinary three layer architecture: a synchronous layer, an asynchronous layer and a mediating queuing layer.

5.1 Static Structures

By comparing the modeling approach presented in [POSA2000] using UML with the FMC approach used in this paper one sees that, despite the different graphical notations, the structures resemble each other. In both cases a hierarchical three layer approach is used to depict the communication of synchronous and asynchronous services via a queuing mechanism (figure 4 and figure 10).

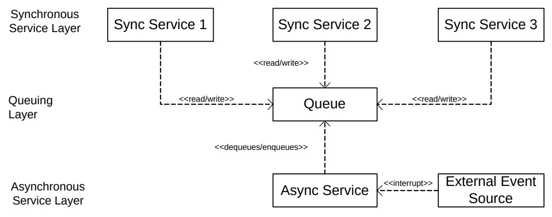


Figure 10 [POSA2000] Class Diagram

Another difference is the way of FMC of modeling the locations where information is stored within the hierarchy of layers. Modeling channels as volatile storage places and the queue as a persistent storage place contributes to the understanding of the pattern's mechanism: information is directly transmitted to the queuing layer where the information is kept until either adjacent partner is available to retrieve the information for further processing. The method used in [POSA2000] based on UML using dependencies does not allow such insights.

5.2 Dynamic Structures

Tightly related to the modeling of the pattern's dynamic structures is a clear definition of synchrony and asynchrony. The description of the pattern in [POSA2000] does not introduce clear semantics of these terms.

Further, it is essential for the pattern's understanding to clearly describe the flow of control during the inter layer communication. In this context a distinction of two use cases representing the two directions of communication is advisable, since the usage of the queuing mechanism - in the role of a sender as well as in the role of a reader - differs between synchronous and asynchronous services.

[POSA2000] describes only one of those use cases. The authors use a UML sequence diagram to model the asynchronous to synchronous direction of communication (figure 11).

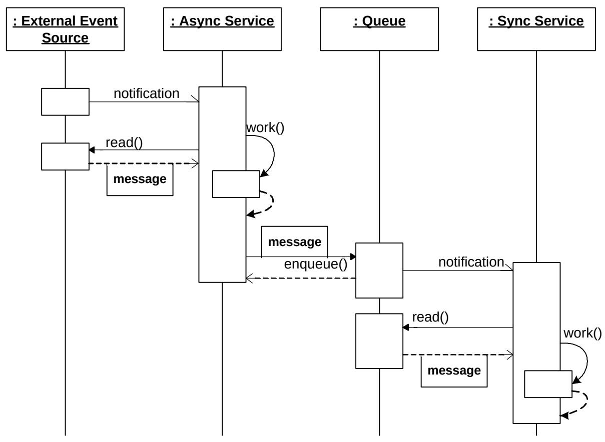


Figure 11 [POSA2000] Sequence Diagram

The sequence diagram does serve well to show exemplary sequences of events but it does not give thorough insight into the topic of exchanging information via the queue. Even though the diagram clearly shows the data flow no control state is shown. Wait states of the synchronous component are not considered either; the places used in the FMC petri net exactly show in which control state a synchronous service waits for an asynchronous service to serve his request. Using

FMC we put emphasis on modeling these wait states. However, the drawback of the FMC petri net technique compared to the UML sequence chart is that a petri net does not model data flows. These, on the other hand, can well be integrated in sequence charts. FMC models data flows using block diagrams which represent static structures. Depending on the readers taste, it can be considered an advantage to include data flows in sequence charts. However, trading in control state information certainly is not an option.

On the other hand, both diagrams have syntactical notations in common to clearly separate concerns. Agents are well separated in both diagrams.

In summary, both techniques fulfil the main task to introduce the pattern. The intention of the pattern is to leave a lot of freedom to the architect or developer, depending on the application domain. Therefore, more restrictions due to increased degree of detail are not desirable. However, the use sequence diagrams combined with the lack of state information in any of the UML diagrams is a major drawback for the [POSA2000] approach which reduces the pattern's comprehension.

6 Comparison to Other Patterns

In general, the Half-Sync/Half-Async Pattern and its variants do not strongly compete with other concurrency patterns. Decoupling the communication of two or more components is a basic requirement which does not allow many alternatives other than the use of queuing mechanisms. The fact that contemporary operating systems always use the Half-Sync/Half-Async pattern when providing asynchronous services supports this statement.

Other patterns which deal with concurrency issues mainly provide a way to organize processes or threads to efficiently serve a special application domain or to minimize performance problems within a given application domain such as event handling. In this context they often use or extend the Half-Sync/Half-Async pattern. Examples for such patterns are the Reactor pattern, the Leader/Follower pattern and the Proactor pattern described in [POSA2000].

Mainly, most other concurrency patterns are more specific than and not as general as the Half-Sync/Half-Async pattern.

7 Conclusion

In this paper, we discussed the application and implementation of the Half-Sync/Half-Async pattern and its variants. We examined it is a concurrency pattern used to decouple communication between synchronous and/or asynchronous participants, therefore using a queuing mechanism.

The method of decoupling communication between synchronous and asynchronous services was well known and implemented several times before the term "pattern" was used in software engineering. Therefore, the Half-Sync/Half-Async pattern does not describe a solution to an unknown problem. However, the fact that the technique has been made a pattern allows developers to understand it to its full extend and with less effort, giving a deeper level of system understanding.

References

[AMP] The Apache Modeling Project, http://apache.hpi.uni-potsdam.de

[POSA2000] D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, Pattern-Oriented Software Architecture. Wiley, 2000

[FMC] Fundamental Modeling Concepts Web Site, http://fmc.hpi.uni-potsdam.de

Leader/Followers

Dennis Klemann, Steffen Schmidt
Hasso-Plattner-Institute for Software Systems Engineering
P.O. Box 90 04 60, 14440 Potsdam, Germany
{dennis.klemann, steffen.schmidt}@hpi.uni-potsdam.de

Abstract

The Leader/Followers pattern provides a high-performance processing of concurrent events with low latency. Its main component is a thread pool consisting of processing, follower and one leader thread. The latter is responsible for the dispatching of an incoming event. Afterwards, it promotes a new leader and starts processing the event. This minimizes locking overhead, prevents failures caused by concurrent access to event sources and works without data exchange between the threads through shared buffers.

Keywords: Patterns, Architecture, Concurrency, Multi-threading, Event handling

1. Introduction

The Leader/Followers pattern provides an architecture for an event-driven , multi-threaded application.

Event-driven means that the application reacts to events, that occur on one or many event sources, by running an event handler. There can be various event handlers. The right one must be chosen depending on the type of the event.

For working efficiently, such an application should be multi-threaded. So, a huge amount of events can be processed simultaneously by running many event handlers concurrently, each one in its own thread.

Well-known examples for this class of applications are all kinds of servers, like web servers or database servers. Here the events are the requests which the clients send to the server. The server reacts to the requests by starting the appropriate event handler that processes the request and sends a response back to the client.

2. Problem

Multi-threading has the advantage that several events can be processed concurrently. But it brings new problems that have to be considered.

2.1. Demultiplexing events

The events can occur on a lot of event sources. But often it is not possible to associate an own listener thread to each single event source, because of scalability limitations of applications or the underlying operating system and network platforms.

Instead of this, the events must be demultiplexed by a small number of threads. A design goal is to find an efficient association between event sources and threads.

2.2. Concurrency-related overhead

Running multiple threads concurrently produces overhead like context switches, synchronization, cache coherency management and inter-process communication. Allocating memory dynamically or creating a new thread for each incoming event produces overhead, too. For efficient event handling all this overhead should be as low as possible.

2.3. Concurrent access to event sources

A multi-threaded application must include a mechanism that prevents the threads from accessing the same event source simultaneously. Otherwise, data could be lost or corrupted or an event could be processed twice. Additionally, this mechanism must assure that the threads don't run into deadlocks.

3. Solution

The Leader/Followers architectural pattern describes an concurrency mechanism that minimizes concurrency-related overhead and prevents simultaneous access to event sources. The events are handled by concurrent tasks, which can be implemented by processes or threads. In the following description threads are used to explain the pattern.

3.1. Structure

The static structure of the pattern consists of two basic elements: handles in a handle set and a pool of threads that share a synchronizer.

A handle identifies an event source and is provided by the operating system. An event source can be a network connection or an open file, for example. The handle to an event source can generate events, like connect-requests or time-outs, and queue them in an internal queue. The handle set is a collection of handles and can be used to wait for the occurrence of an event on any of the handles. It returns to its caller when it is possible to initiate an operation on a handle in the set without the operation blocking.

The thread pool is a collection of a fixed number of threads. The threads are created during the initialization phase of the application and are not terminated until the shut-down of the application. The number of threads can be adjusted for load balancing only. The threads are responsible for detecting, demultiplexing, dispatching and handling of the events. Therefore the threads play three different roles in turn:

- *Leader*: waiting for a new event in the handle set and demultiplexing it
- *Processing*: handling an event by running an event handler
- *Follower*: waiting to play the leader role

While there can be multiple followers and processing-threads, there is at most one leader thread in the thread pool. For coordinating the roles of the threads, the thread pool contains a synchronizer, e.g. a semaphore or a mutex. The follower threads queue up on the synchronizer and wait to become the leader.

The processing threads dispatch the events to event handlers, which implement a specific service that is offered by the application. They are started by calling a hook method in reaction to the occurrence of an event and run in the context of a processing thread.

Figure 1 shows the compositional structure described above. It must be mentioned, that this block diagram visualizes the structure at a certain point of time. Because the threads change their roles, the numbers of threads in each role differ. It is possible that there is temporary no leader, no follower or no processing thread.

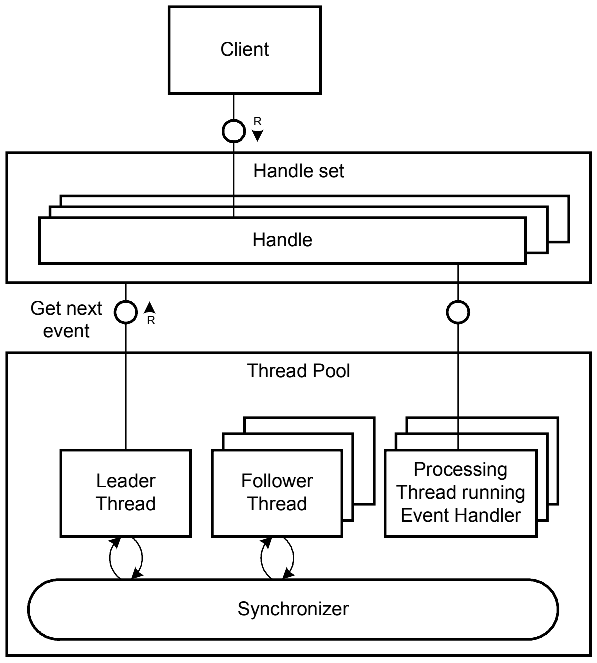


Figure 1 block diagram showing the static compositional structure

3.2. Dynamics

By playing different roles, all steps for processing an event (detecting, demultiplexing, dispatching, handling) can be done by the same thread. Therefore context switches and data exchanges between threads are not necessary. Figure 2 shows the connections between the roles and when a thread changes its role.

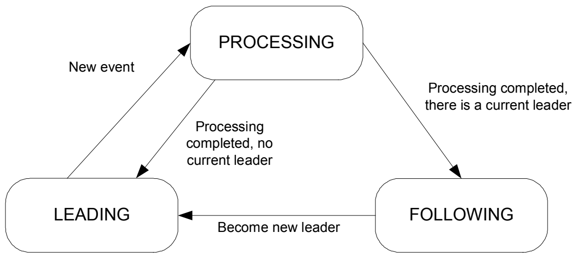

PROCESSING
New event
Processing completed, there is a current leader
Processing completed, no current leader
LEADING
Become new leader
FOLLOWING

Figure 2 Transitions between the roles

3.2.1. The leader role

The current leader thread is the only thread in the thread pool which detects and demultiplexes new events. Therefore it waits for the occurrence of an event on any of the handles in the handle set. If there is no current leader thread, because all threads are in the processing role, incoming events are queued up by the operating system until a leader is available.

After detecting an event, the handle on which the event occurred must be deactivated. So, it is not possible that the next leader tries to access the handle and to demultiplex the same event again.

The final step of the leader thread is to promote a new leader. One of follower threads is chosen to become the next leader thread that waits for new events. The follower promotion protocol can be implemented in different ways:

- *LIFO order*: The thread with the shortest waiting time is promoted first. By promoting the threads in last-in, first-out order, the CPU cache affinity can be maximized. This improves the performance of the system but requires an additional data structure, that holds the order of the follower threads.
- *Priority order*: If the threads run at different priorities, it is useful to promote the follower threads according to their priority. So the effect of priority inversion can be minimized. This ordering should be used for real-time applications. A priority queue is required to find the follower thread that has to be promoted.
- *Implementation-defined order*: The easiest and most common way is to use synchronizers provided by the operation system like semaphores, mutexes or condition variables. The follower thread to promote is selected by the operating system specific implementation of the synchronizer. The use of native operating system synchronizers is very efficient.

If all other threads are in the processing role, no follower is available. In this case the old leader changes to processing role without promoting a new leader. The leader role remains vacant until any of the processing threads has finished event handling.

3.2.2. The processing role

After a new leader thread is found, the event can be dispatched to an event handler. The thread that detected the event now plays the role of a processing thread. It selects the appropriate event handler and starts event handling by calling the hook method. The event handler runs in the context of the processing thread. So it can execute concurrently with other processing threads and the leader thread.

When event handling is finished, the handle is reactivated, so that new events arriving on it can be demultiplexed.

3.2.3. The follower role

A thread that has completed event handling can try to become the leader thread again. If there is no current leader, it can play the leader role immediately. Otherwise the thread must wait for the synchronizer as a follower until it is promoted.

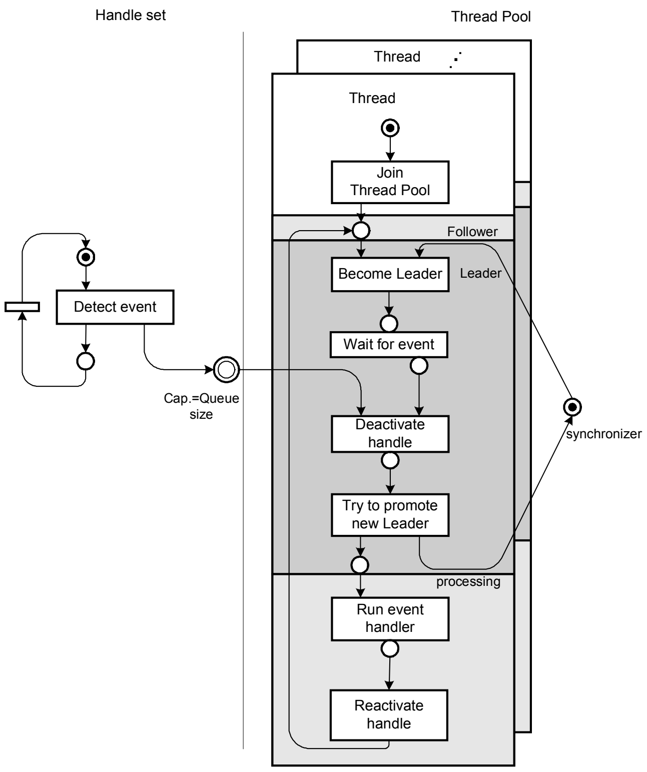


Figure 3 Petri net showing the dynamic structures (with roles)

4. Relationship to other patterns

The Leader/Followers pattern describes an event handling mechanism. Other patterns that deal with event handling are the *Reactor* pattern and the *Half-Sync/Half-Async* pattern.

The Reactor pattern uses a single thread performing an event loop that listens for events. After one or multiple events occurred, they are dispatched to an event handler. If the event handler doesn't run in an own thread, it is not possible to process multiple events concurrently. So performance can be very low if a huge amount of events must be processed. The performance can be improved by running the event handlers in own threads. But this produces overhead by dynamically creating the threads and performing context switches. So the Reactor pattern should be used instead of the Leader/Followers pattern if there are only few events and the time to process an event is short.

The main benefit of the Reactor pattern is its mechanism to dispatch events to the appropriate event handlers. The event handlers can register their handles with the reactor and are informed if an event occurs on these handles. This can be used in the Leader/Followers pattern. The leader thread can demultiplex and dispatch events by implementing the Reactor pattern.

The Half-Sync/Half-Async pattern distributes event handling to synchronous and asynchronous services. The service that detects events passes a message to the processing service over a queuing layer. In contrast, handling of a single event is done synchronously by one thread in the Leader/Followers pattern. That produces less synchronization overhead and a queuing layer is not necessary.

The Half-Sync/Half-Async pattern should be used instead of the Leader/Followers pattern if there are additional synchronization or ordering constraints that must be addressed by reordering requests in a queue before processing them or if event sources cannot be waited for by a single event demultiplexer efficiently.

5. Variants of the pattern

There are a couple of variants of the Leader/Followers pattern. Some make use of multiple leaders, others bind handles to a specific thread.

5.1. Multiple leaders

In contrast to the standard pattern, in this case there are several leader threads waiting for events. Multiple leaders are necessary if the handle set is divided into a certain amount of subsets. This can occur when there are multiple event sources like I/O, semaphore and/or message queue events, as e.g. UNIX provides no multiplexing function that can wait for those sources simultaneously.

5.1.1. Relaxing Serialization Constraints

Some operating systems provide functions for multiple threads to wait for a single handle set, e.g. Win32's WaitForMultipleObjects function, which notifies only one waiting thread if an event occurs. In this case, one can take advantage of multi-processor hardware.

5.1.2. Leaders/Followers per handle subset

In this variant, every handle subset gets assigned a thread pool with one leader and a certain amount of follower threads. Thus, each thread is limited to a specific handle set.

5.1.3. Multiple leaders and multiple followers

Like the aforementioned variant, there are (maximally) as many leader threads as handle subsets. The difference is that there is only a single thread pool. In consequence, all threads can be used for any handle. When an event occurs, the relevant leader promotes the new leader for this handle subset. After having processed an event, a thread checks if all handle subsets have leaders assigned to them. If not, it gets promoted to leader immediately. Otherwise, it becomes a follower.

5.2. Bound handle/thread association

In some cases is it useful to assign a specific thread to a handle, for example a front-end server. If there is a request from a client which needs a back-end server, the thread which processed this event is well suited to handle the response from the server and answer the client, as this thread still possesses the needed context information. In an implementation using this variant, when an event occurs, the leader thread checks if it is responsible for this event. If this is the case, there is no difference to the standard pattern. If not, the leader thread hands that event off to the responsible thread and waits for the next event. The responsible thread directly processes the event, skipping the leader role.

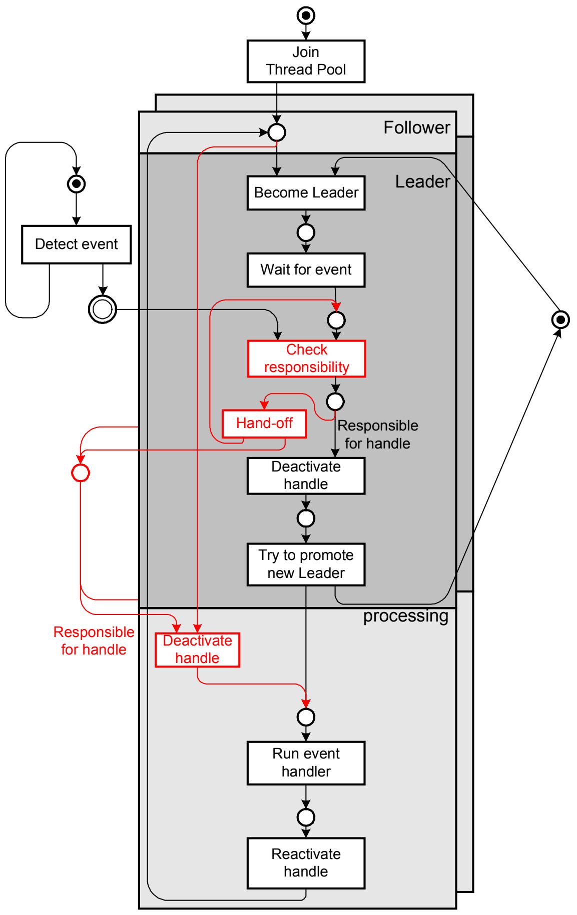


Figure 4 Additions to the petri net for the bound handle/thread association

6. Use Cases

As a high performance web server, the Apache web server needs to process a lot of requests. It makes use of the Leader/Followers pattern to fulfill this requirement.

In Apache 1.3, the so-called "Request-Response-Loop" [1] is responsible to handle the client requests. In the follower role, the preforked child processes try to acquire an accept mutex, which is equivalent to the follower role. Once a request comes in, one of the threads is promoted leader, demultiplexes the event and processes it. This means that the operating system chooses the next thread to become leader. After having processed the request, the thread waits until it can access the accept mutex again.

Apache 2 provides 2 MPMs (Multi-Processing Modules) that make use of this pattern. The first one is the Prefork MPM, which behaves just like the Apache 1.3. The other one is the Leader MPM, still in experimental state, which implements the pattern using a follower stack for leader promotion. In that way, the followers get chosen in LIFO order, making use of cache affinity, as it is most likely that the new leader thread is still in the cache. This eliminates the need to get the thread and its context from memory, saving time and memory bandwidth.

7. Related work

One of the most detailed descriptions of the Leader/Followers pattern is written by Douglas Schmidt et al. [2].

While the verbal description is very accurate and insightful, the diagrams left something to be desired. Some are not as comprehensive as they could be, some are faulty – e.g. the state chart. Figure 2 of this article shows the correct transitions.

The complexity of the sequence diagram indicates the difficulty in visualizing the dynamics of multithreaded systems. One has to look very closely to understand the dynamic structure of this system – and still some questions stay. For example, the time span of the event processing is not shown.

In our opinion, a petri net is more suitable to illustrate the behavior. Not only does it provide a clearer look at the dynamics – due to the fact that it has a very strict interpretation –, but also has the advantage of a general flow over an exemplary flow of the sequence diagram.

In addition, we made a slight change to the behavior of the pattern inasmuch the thread does not leave the thread pool after completion of the processing role, as we saw no real advantage of that design decision.

8. Conclusion

The Leader/Followers pattern is useful in a multi-threaded, event-driven environment which is required to handle a lot of request simultaneously and with minimal latency.

This is due to the fact that no data has to be exchanged between the threads, as the thread dispatching the event is the very same that processes it. This minimizes locking overhead and makes shared buffers obsolete. When the follower get promoted in LIFO order, cache affinity is another important factor for a system that meets the abovementioned requirements.

Usually, there is a fixed amount of threads and only one leader at maximum, though a variant with more leaders can occasionally be more efficient.

Bibliography

[1] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, Oliver Schmidt (2003) *The Apache Modelling Project*. Forschungsprojekt am Hasso-Plattner-Institut Potsdam. http://apache.hpi.uni-potsdam.de/document/the_apache_modelling_project.pdf

[2] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture Volume2 – Patterns for Concurrent and Networked Objects*. Chichester: Wiley