Loose Coupling and Architectural Implications

Keynote, International Conference on Web Intelligence (WI), Leipzig – Germany, August 23 – 27, 2017

Prof. Dr. Dr. h.c. Frank Leymann
Member, The Academy of Europe
Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstr. 38
70569 Stuttgart
Germany

Phone +49-711-685 88 470
Fax +49-711-685 88 472
e-mail Leymann@iaas.uni-stuttgart.de
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Microservices
Cloud
Pattern Languages
Conclusion
Integration Problem: Tight Coupling is Problematic

Application Architecture of a real company 😊
RPC (Remote Procedure Call)

- Attempt to make remote communication via RPC behave like local call
- **Upside:**
  - Semantics familiar to developers
- **Downside:**
  - Remote (!) communication invalidates many assumptions local calls are based on
  - Thus: RPC simply does **not** have semantics of a local call!
RPC ≠ Local Call (1/2)

- Called program might not be available (network fragmentation, server outage, ...)
  ⇒ Time Dependency
    - How long should caller wait?

- Remote program often controlled by a different party (business partner, ...)
  ⇒ Format Dependency
    - What if signature of called program changes?
      - Rewrite code of calling program! 😞
Calling program often explicitly points to a particular target program
⇒ Reference Dependency
- What if called program is moved to different machine?

- Parameters have to match machine architecture (little/big Endian,...), locale settings (code pages, time zones,...)
⇒ Platform Dependency
\[ \Sigma = \text{Causes of Volatility of RPC} \]

- **Time Autonomy violation**
  \( \approx \) All ingredients have to be available at same time

- **Format Autonomy violation**
  \( \approx \) Number and types of parameters must match

- **Reference Autonomy violation**
  \( \approx \) Hard-coded addresses

- **Platform Autonomy violation**
  \( \approx \) Internal representations of data (little/big endian...)
How to Loosen Coupling 1-by-1

- Reference: Send information to a logical address ➔ Channel
  - Channel is logical address sender and receiver agree on

- Time: Enhance channel with queuing capabilities
  - Request/response wait until network or recipient is available

- Formats: Enhance channel with format transformation (aka mediation) capabilities

- Platform: Use standards for message structures and choose supporting software ubiquitously available
Finally: Definition of Loose Coupling

- **Reference Autonomy**
  - Caller and callee don’t know each other

- **Time Autonomy**
  - Caller and callee execute at their own pace

- **Format Autonomy**
  - Caller and callee may use different formats of data exchanged

- **Platform Autonomy**
  - Caller and callee may be in different environments, written in different languages, ...
Producers and consumers communicate via queues or topics, i.e. they don’t know each other.

Routing components free producers even from having to know which queue or topic to use:

...i.e. producers only has to understand single source to send messages to.
Asynchrony is at the heart of messaging, thus time autonomy is “in its guts”

Queuing is providing 1-1 connections

Pubsub is providing 1-n connections
Platform Autonomy in Messaging

- Endpoints are isolating applications from messaging environment
- Application can be...
  - ...in any programming language
  - ...in any operating environment
  - ...
- Messaging products run in many environments
- You can even bridge between different products
- Transformation of messages is done “on the wire”
- Versioning of messages is enabled
- ...

Format Autonomy in Messaging
Agenda

1. Loose Coupling Messaging
2. WS*
3. REST
4. High Availability
5. Scalability
6. Stability
7. Microservices
8. Cloud
9. Pattern Languages
10. Conclusion
Message Endpoints

- Applications typically do not send data to or receive data from a channel
  - They are not built with integration in mind!
- **Message endpoints** connect applications to integration solution
Characteristics of a Message Endpoint

...is reachable via several transports & message encodings

...is at an address

...implements an interface

Binding

Port

WSDL
- Message(s) received or sent “stimulates” another message(s) (result, fault, ...)
- Grouping messages for describing their relation and semantics is needed:
  ➔ Operation
- Typically, a message processing endpoint can cope with multiple groupings of messages
- Grouping of operations for describing their relation and semantics is needed:
  ➔ Port Type
WSDL – Reachability

- A concrete endpoint (i.e. message processor) implements a particular port type:
  ➔ Port

- A port...
  - ...is reachable via a transport protocol
  - ...supports a serialization format
  ➔ Binding

- A message processor may support different ports (i.e. may be reachable via multiple bindings, multiple formats,...):
  ➔ Service
Loose Coupling in WS*

- **Reference Autonomy**
  - Client interacts with an endpoint/port not with a concrete program, i.e. client doesn’t know the actual service

- **Time Autonomy**
  - Asynchronous bindings (JMS,...) can be used
  - Reply-to header, Correlation header,... support asynchrony even over synchronous transports

- **Platform Autonomy**
  - Client interacts with service without having to know programming language, hosting environment,... of service

- **Format Autonomy**
  - Binding specifies serialization format
  - Transformation can be done “invisibly” along the wire
Core Idea Behind REST

- REST is an architectural style

Key Points:
- **RE**presentational
- **State** Transfer
Characteristics Of The Web Architecture

- Interaction with **URI** addressable resources
- Fixed set of **generic interactions**
  (mainly HTTP GET, POST, PUT, and DELETE)
- **Standard data format** (HTML, XML,... - MIME types)
- **Stateless**
- All context is fully understandable from message
  (→ “Visibility”)
Characteristics Of The Web Architecture (cont.)

- State of interaction is incorporated in message returned to caller, not maintained on server → Scalability & recoverability from servers’ perspective!
  - ≡ “State Transfer”
  - This avoids “server affinity”
    - I.e. server can be build as cluster (server farm, cloud, ...)

- Navigation & formatting options negotiated between client and server
  - ≡ “Representational”

- Large grained interactions
  - Large grained information retrieval to get resources to caller
  - Small to medium grained interactions for query and control information
  - ≡ “Fixed Set of Generic Interactions”

- Simple security model based on ACLs based on URI
  - Well, this became more sophisticated (OAuth, OIDC, ...)
- URI decouples from knowing the actual function implementation
- HATEOAS goes beyond
Time Autonomy in REST: Long Running Requests

- Sometimes requests may take too long for a client to wait for response
  - Especially POST and DELETE may take some time
    - They might initiate a complete business processes

- Server acknowledges reception of request
  - Via HTTP Status Code 202 Accepted

- Server provides a resource that client can use to track progress of long-running request (aka “task”)
  - Via Content-Location header of this resource
Example: Starting Long Running Request

POST /imageProcessing/beautify HTTP/1.1
Host: www.fl.com
Content-Type: multipart/related; boundary=fl42

--fl42
...
--fl42
...
--fl42--

HTTP/1.1 202 Accepted
Content-Type: application/xml
Content-Location: http://www.fl.com/imageProcessing/tasks/1

<status>
  <state>running</state>
  <link rel="self" href=".../tasks/1"/>
  <estimatedCompletion>2020-04-01</estimatedCompletion>
</status>
Example: Checking State of Long Running Request

GET /imageProcessing/tasks/1 HTTP/1.1
Host: www.fl.com

HTTP/1.1 200 OK
Content-Type: application/xml
Content-Location: http://www.fl.com/imageProcessing/tasks/1

<status>
  <state>running</state>
  <link rel="self" href="/tasks/1"/>
  <estimatedCompletion>2020-10-10</estimatedCompletion>
</status>
Example: Completion of Long Running Request

GET /imageProcessing/tasks/1 HTTP/1.1
Host: www.fl.com

HTTP/1.1 303 See Other
Content-Type: application/xml
Location: http://www.fl.com/images/314
Content-Location: http://www.fl.com/imageProcessing/tasks/1

<status>
    <state>ready</state>
    <link rel="self" href=".../tasks/1"/>
    <message>Image processed & stored</message>
</status>

- **Location** header points to newly created, i.e. successfully processed resource but not to the task resource
- **Content-Location** header (still) points to the task resource
Example: Failed Long Running Request

```
GET /imageProcessing/tasks/1 HTTP/1.1
Host: www.fl.com

HTTP/1.1 200 OK
Content-Type: application/xml
Content-Location: http://www.fl.com/imageProcessing/tasks/1

<status>
  <state>failed</state>
  <link rel="self" href=".../tasks/1"/>
  <message>Image could not be processed</message>
</status>
```
Platform Autonomy in REST

- The premier transport to implement REST is HTTP(S)
- HTTP clients and HTTP servers are available in all major environments
- Programs in most languages can be producers and consumers of HTTP messages
Often, requested entity is available in multiple variants (≈ formats)

HTTP defines procedures to negotiate the appropriate variant to be transferred

Client may send input in various formats
RESTful Web Services vs. “Big” Web Services: Making the Right Architectural Decision

Cesare Pautasso
Faculty of Informatics
University of Lugano
via Buffi 13
6900 Lugano, Switzerland
cesare.pautasso@unisi.ch

Olaf Zimmermann
IBM Zurich Research Lab
Saeumerstrasse 4
8803 Rueschlikon, Switzerland
olz@zurich.ibm.com

Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart, Germany
frank.leymann@iaas.uni-stuttgart.de

ACM 978-1-60558-085-2/08/04.
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Microservices
Cloud
Pattern Languages
Conclusion
Two Classes of Faults

- **Fault** is an event within a system that caused a failure

- **Transient** fault does not re-occur when retrying the operation ("Heisenbug")

- **Permanent** fault is repeatable ("Bohrbug")
Improving Availability of Software

If software fault is a Bohrbug there is no repair:

- Get an emergency fix or wait for the next release

If software fault is a Heisenbug then repair is easy:

→ Shutdown and restart

Luckily, most software-caused faults are Heisenbugs
A **watchdog** is a special program that detects faults of application components fast.

It does so by...

- Monitoring the liveness of these components, and
- Re-creating failed components “immediately”

...which is equivalent to shutdown and retry 😶

⇒ Good for fixing Heisenbugs 😊
Watchdogs: Fault-Detection Monitoring

- Queue Cardinality
- Lock Acquisition
- Pulse
- Heartbeat
Hot-Pooling: High Availability

- **Hot Pool**: A collection of application components with identical functionality sharing a common input queue
- When a member fails the other members will continue processing requests from input queue ⇒ Availability
- Watchdog will restart failed members ⇒ High Availability

![Diagram of Hot Pooling](image)

- Client
- PUT
- GET
- App
- Update
- Watchdog
- Hot Pool

= Shutdown & Retry
A Bit of Math: Forgive Me 😊

\[
\alpha_{\text{hot pool}} = P_{\geq 1} = \sum_{i=1}^{N} \binom{N}{i} \alpha^i (1-\alpha)^{N-i} = \sum_{i=0}^{N} \binom{N}{i} \alpha^i (1-\alpha)^{N-i} - \left( \binom{N}{0} \alpha^0 (1-\alpha)^N \right) = (\alpha + (1-\alpha))^N - (1-\alpha)^N
\]

...a lot of explanations, e.g. application components fail independently; independent events are “binomial distributed”; availability means that at least one member is running in hot pool; etc etc...

\[
\Rightarrow \alpha_{\text{hot pool}} = 1 - \left( 1 - \alpha_{\text{member}} \right)^N
\]

Example: Assume a terribly bad application component with 80% availability (\(\alpha_{\text{member}}=0.8\)).

\[
\Rightarrow \text{Hot pool with 8 of such bad components has 99.999 availability (\(\alpha_{\text{hot pool}}=0.99999\))!}
\]
Underlying Hot Pools...

...is loose coupling!

Client

PUT

GET

App

Update

© Frank Leymann
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Microservices
Cloud
Pattern Languages
Conclusion
Hot-Pooling: Scalability & Elasticity

- Watchdog can monitor metrics of SLOs of hot pool
- If SLOs are threatened to be missed, watchdog will start additional members ⇒ Scale-out
- If members are underused, watchdog will shutdown subset of members ⇒ Scale-in

Elasticity: Automatic scale-in/out
Clusters: Increasing HA & Scale

An **Cluster** is a “hot pool of hot pools” on different machines.

The cluster provides a **single system image**, i.e. clients submit requests to the cluster **not** to a hot pool on a particular machine.

(via cluster-wide queue ⇒ loose coupling)
An *Application System* is the complete, interdependent set of hardware, middleware, application components,... required to process complete business requests.
An application system (or system for short) is called **stable** iff it keeps processing business requests even when there are transient impulses, persistent stresses, or component failures disrupting normal processing.

- **Impulse** is a rapid shock to a system.
- It's like whacking the system with a hammer.

- **Stress** is like a force being applied to a system over an extended period of time.
- It's like constantly pushing against the system.
Impulse in Software

- 100,000 new sessions within a minute is an impulse
- 8,000,000 new messages in a queue within a few minutes before close of business is an impulse

- Such an impulse can break a system within seconds!
Slow response from your credit card processor over a long period of time is stress

...because it reduces the capacity to serve customers

**Stress produces strain**

Stress from the credit card processor produces strain that propagate to other parts of the system

...resulting in requests piling up in queues, queues getting full, refusing new requests, stopping services to customers
Long Period of Time

- How long is a "long period of time"?

  It depends!

- A very good definition is the time between two code deployments

- If you deploy your system once a week it doesn't matter to build the system to run for five years without interrupt
Sudden impulses and excessive strain can both trigger catastrophic failure!

Analogy:
a steel plate with a microscopic crack in the metal. Under stress the crack can begin to propagate faster and faster until the plate brakes!
Example: automotive engineers build crumple zones into the car body to protect passengers by failing first.

Similarly, build your components to stop cracks propagating into the rest of the system.
Tight Coupling Propagates Cracks!

Order Entry \(\rightarrow\) Sync Call \(\rightarrow\) Credit Card Processing

© Frank Leymann
Stop Propagating Cracks!

Queues are "crumple zones"!
The Role of Loose Coupling

Loosely coupled components act as shock absorber, diminishing effects of errors instead of amplifying them

- amplification is typically done by tight coupling:
  
  **tight coupling accelerates cracks**
Intermediate Conclusion

Loose coupling supports building highly available, scalable, elastic and stable components in a straightforward manner.
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Microservices
Cloud
Pattern Languages
Conclusion
...microservice architectural style is an approach to
- developing a single application as a suite of small services,
- each running in its own process and
- communicating with lightweight mechanisms, often an HTTP resource API.

These services are built
- around business capabilities and
- independently deployable by fully automated deployment machinery.

There is a bare minimum of centralized management of these services, which
- may be written in different programming languages and
- use different data storage technologies.

(*) J. Lewis & M. Fowler: “Microservices” (2015),
http://martinfowler.com/articles/microservices.html
Microservice: Main Properties

Microservice is:
- small
- running in its own process
- communicating often [via] HTTP
- built around business capabilities
- written in different programming languages
- use different data storage technologies
- independently deployable by fully automated deployment machinery

...what ever that means 😞

True for many (!) service

True for many (!) service, and most REST services

That’s what services are all about!

That’s interesting 😊
A microservice is a service that is independently deployable by fully automated deployment machinery...

...and this has interesting implications (see next)
You have a “big application” (a.k.a. “monolith”)

If you want to replicate it
  - e.g. because of scale, availability, elasticity,...

you pay penalties (e.g. licenses for all components)
although you only need “a few components” replicated
Split your “big application” into smaller granules that can be deployed independently
- I.e. split it into microservices

Replication can be restricted to those components that turn out to be bottlenecks (⇒ scale), critical (⇒ HA),... without paying the penalties for all other components of the “big application”
Microservice-Based Systems are Reactive(*)

- **Elastic**
  - ...they are scalable

- **Responsive**
  - ...they respond in a timely manner, often non-blocking

- **Resilient**
  - ...they isolate failures, avoiding their cascading

- **Message-based**
  - ...queues are crumple zones
  - ...queues mean asynchrony
  - ...queues enable hot pools

Loose coupling is the basis for building reactive systems, especially microservices

(*) http://www.reactivemanifesto.org/
Agenda

- Loose Coupling
- Messaging
- WS*
- REST
- High Availability
- Scalability
- Stability
- Microservices
- Cloud
- Pattern Languages
- Conclusion
IDEAL Principle: Native Cloud Application Properties - Architects View

**Isolated State:** most of the application is *stateless* with respect to...
*Session State:* state of the communication with the application

**Distribution:** applications are decomposed to...
... use multiple cloud resources
... support the fact that clouds are large globally distributed systems

**Elasticity:** applications can be scaled-out and scaled-in dynamically

**Automated Management:** runtime tasks have to be handled quickly
Example: automatic license acquisition because of use of additional resources

**Loose Coupling:** influence of application components is limited
Example: failures should not impact other components
Example: addition / removal of components is simplified
Pattern Languages: Abstraction

P_1

P_6

P_7

P_5

P_3

P_4

P_2

...
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Microservices
Cloud

Pattern Languages

Conclusion
Sample Application

Client → Server

Reliable

Scalable

High Available
The “Reliable” Solution Path

Scalable

High Available

Watchdog

Load Balancer

Hot Pool

N-Plexing

Reliable

Queuing

Guaranteed

tRPC

AtMostOnce

© Frank Leymann
Solution Paths: Abstraction

P₁

P₂

P₃

P₄

P₅

P₆

P₇
The Impact of the Solution Path on the Application Architecture

- Reliable
- Scalable
- High Available
The “Scalable” Solution Path

- Scalable
- Load Balancer
- Hot Pool
- Queuing
- Reliable
- tRPC
- AtMostOnce
- Guaranteed
- High Available
- N-Plexing
- Watchdog
The Impact of the Solution Path on the Application Architecture

- Client
- Server
- Scalable
- High Available
The “HA” Solution Path

- Scalable
  - $\infty$

- High Available
  - ✔
  - High Available

- Load Balancer
- Hot Pool
- N-Plexing
- Guarenteed

- Reliable
  - $\bigcirc$

- Queuing

- tRPC
  - !

- AtMostOnce
- $X$

© Frank Leymann
The Impact of the Solution Path on the Application Architecture

#Listeners=?
Multi-Layer Solution Paths
Pattern Creation Looses Solution Knowledge

Observation
Creation ...

(Solution-) Knowledge

analyze, abstract, evaluate

Pattern Language

Problem

search, navigate

{Pattern}

Solution

implement

Lost knowledge! 😞
Associate Patterns with Solutions

\[ S_1 \]

\[ (c_{11}, \ldots, c_{1n(1)}) \]

\[ S_2 \]

\[ (c_{21}, \ldots, c_{2n(2)}) \]

\[ \ldots \]

\[ S_m \]

\[ (c_{m1}, \ldots, c_{mn(m)}) \]
Pattern Navigation and Solution Aggregation

Solution Path

Aggregated Solution

(c, ...)

(c', ...)

???
Sample Aggregation

sc={WAR will be deployed on Azure,...}
sc={WAR will be deployed on Beanstalk,...}
sc={WAR will be scaled}
sc={PHP will be scaled,...}

precondition: WAR file
postcondition: WAR on Azure
precondition: WAR file
postcondition: WAR on Elastic Beanstalk
precondition: WAR on Elastic Beanstalk
postcondition: WAR is scaled

✔

(s,g,...) ⊕
Agenda

Loose Coupling
Messaging
WS*
REST
High Availability
Scalability
Stability
Cloud
Microservices
Conclusion
Summary

- Building robust systems is all about proper architecture
- Loose-Coupling is a central principle of such architectures
- Loose-Coupling is supported by mainstream architectural styles & technology stacks
- Loose-Coupling is at the heart of microservices
- Pattern Languages help to apply corresponding best practices to result in proper architecture
- Combination of patterns and associated solutions is a sample open research problem
Thanks for your attention!