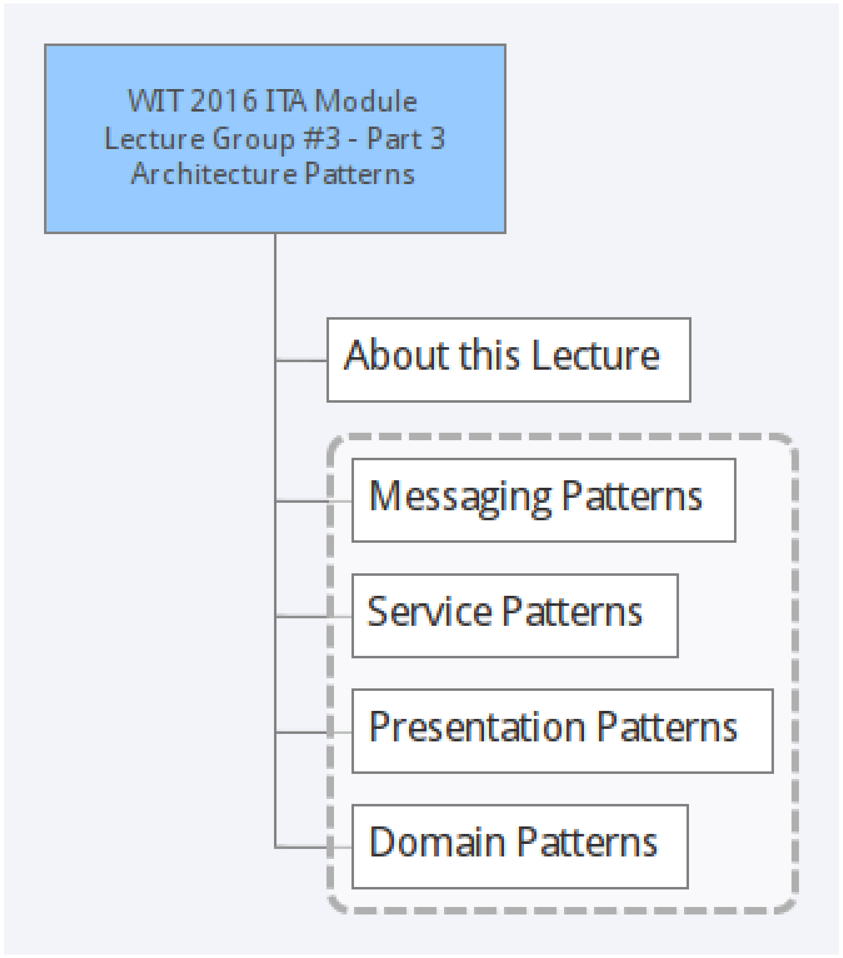
WIT 2016 ITA Module

Architecture Patterns
Lecture Group #3 - Part 3

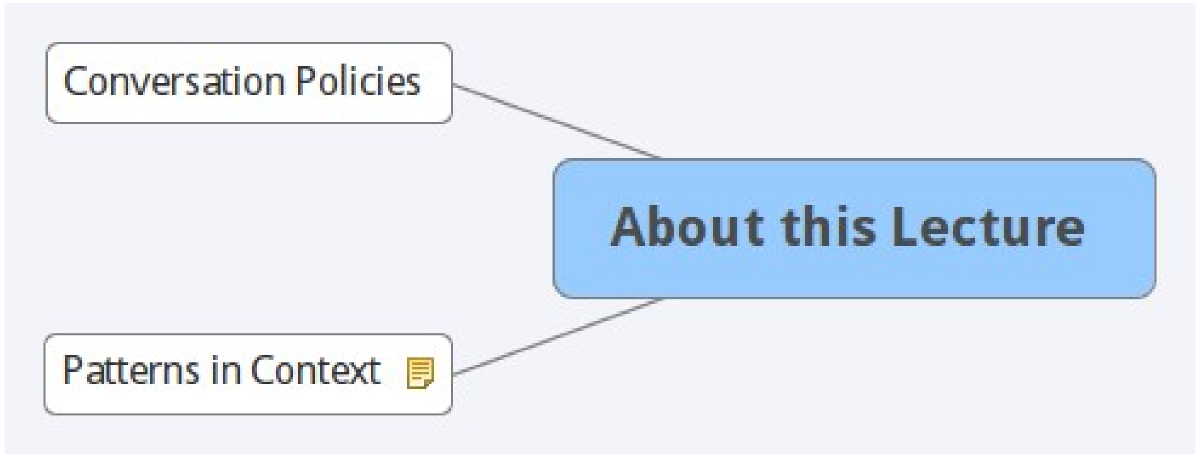
(cc) BY-NC-SA

Lecture Group #3 - Part 3
Architecture Patterns



BY-NC-SA

About this Lecture



(cc) BY-NC-SA

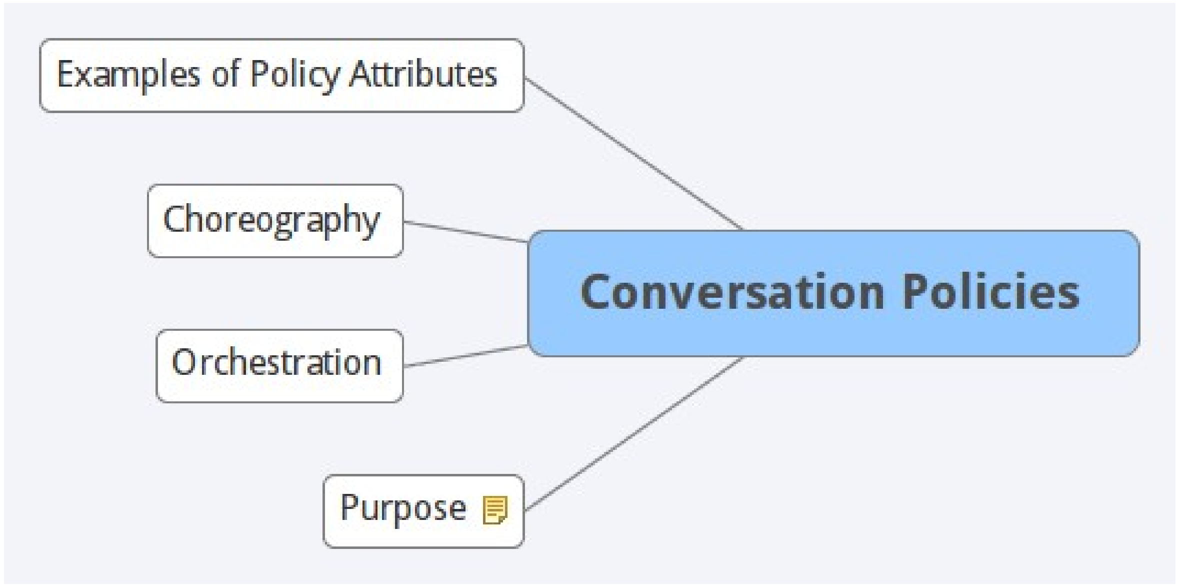
Patterns in Context

The list of architecture patterns presented in this Lecture pertain to remote interactions/collaborations between distributed components.

Architecture Patterns that pertain to intra-communications WITHIN a same component and do not involve distributed components are not the focus of this Lecture.

BY-NC-SA

Conversation Policies



Purpose

An architect must define the CONVERSATION POLICY of an architecture BEFORE choosing the technologies, protocols, delivery mechanisms, types of components of an architecture.

The design of conversations is often overlooked by technical designers and developers to the profit of a choice of technology or framework. Architects have a responsibility to take a step back and state conversations policies between types of components independently from implementation choices.

(cc) BY-NC-SA

What is a conversation?

Discussion Point: Considering a given set of E-mail exchanges happening between a set of team members sharing a same objective.

- What is the "Command and Control" people management paradigm?

- What is the "Delegation and Empowerment" people management paradigm?

What are the pros/cons of each of these paradigms?

(cc) BY-NC-SA

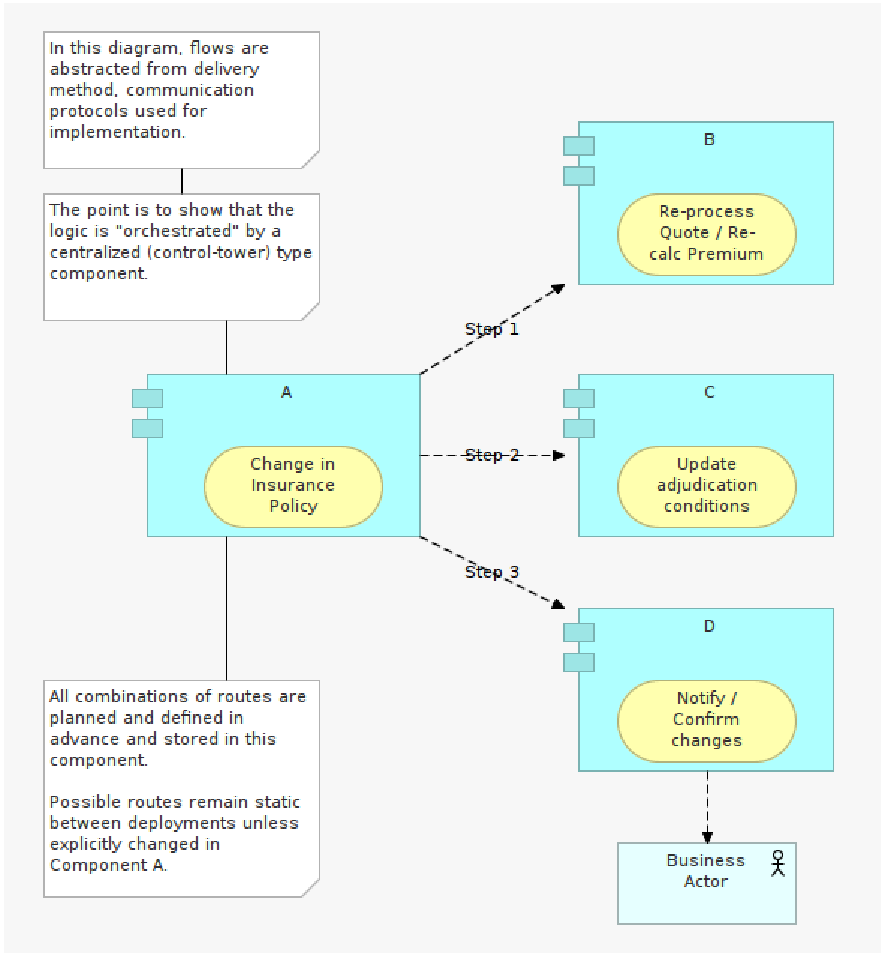
Definition

A conversation policy is a directed exchange of related messages synchronously or asynchronously, over time, between distributed components.

A conversation policy defines what happens between communicating parties — that is, who is allowed (or expected) to send messages to whom, where, why and in what order.

(cc) BY-NC-SA

Orchestration Overview


In this diagram, flows are abstracted from delivery method, communication protocols used for implementation.
The point is to show that the logic is "orchestrated" by a centralized (control-tower) type component.
All combinations of routes are planned and defined in advance and stored in this component.
Possible routes remain static between deployments unless explicitly changed in Component A.
A
Change in Insurance Policy
Step 1
Step 2
Step 3
B
Re-process Quote / Re-calc Premium
C
Update adjudication conditions
D
Notify / Confirm changes
Business Actor

(cc) BY-NC-SA

Paradigm

Orchestration is independent from the choice of RPC communication protocols or methods of MESSAGE delivery between distributed components.

It is a conversation paradigm, and can be implement in many different ways.

Using Message-Oriented patterns, when messages are always transiting through a centralized component before deciding of a next step in logic, it is a form of orchestration.

Using Service-Oriented patterns, when operations implementing a sequential logic flow are invoked from a centralized component, it is a form of orchestration.

As long as the RESPONSIBILITY for the sequencing of steps is CENTRALIZED in a dedicated component: it is a form orchestration.

Whether the decision of next step in sequence is performed statically, based on a planned route, or dynamically, based on the contents of messages exchanged; as long as the orchestration is centralized, it is a form of orchestration.

The conversation is always centralized.

(cc) BY-NC-SA

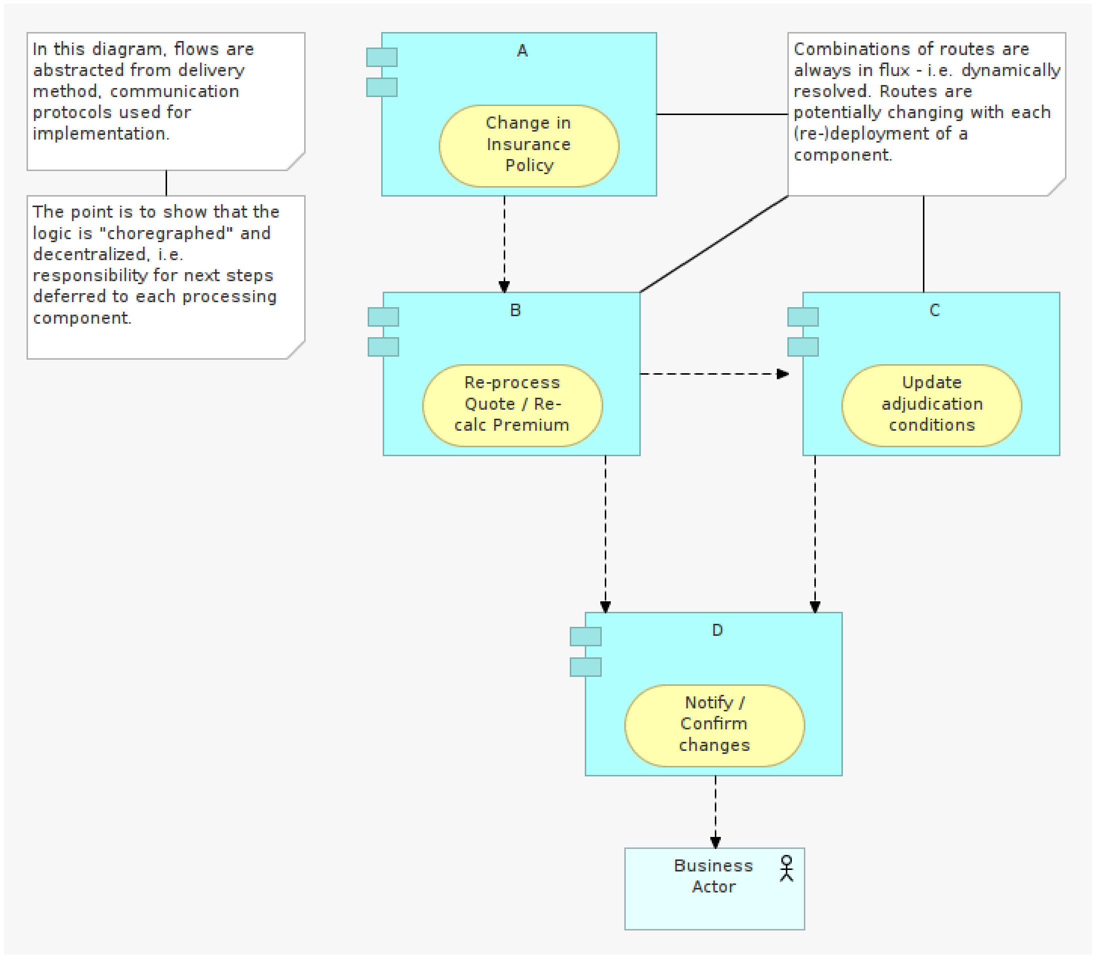
Example

On reception of a Message, the current CONVERSATION STATE is recovered from a set of correlation identifiers, and used to determine the next CONVERSATION ROUTE (i.e. and execute the next step by sending a follow-on Message the next component).

Relevant standard: WS-BPEL

cc BY-NC-SA

Choreography Overview



In this diagram, flows are abstracted from delivery method, communication protocols used for implementation.
The point is to show that the logic is "choregraphed" and decentralized, i.e. responsibility for next steps deferred to each processing component.
A
Change in Insurance Policy
Combinations of routes are always in flux - i.e. dynamically resolved. Routes are potentially changing with each (re-)deployment of a component.
B
Re-process Quote / Re-calc Premium
C
Update adjudication conditions
D
Notify / Confirm changes
Business Actor

(cc) BY-NC-SA

Examples

Choreography is is a ALSO conversation paradigm, but can only be implement using Message-Oriented patterns.

Messages transferred between distributed components contain all the information required for the target component to decide on what next step in logic to execute.

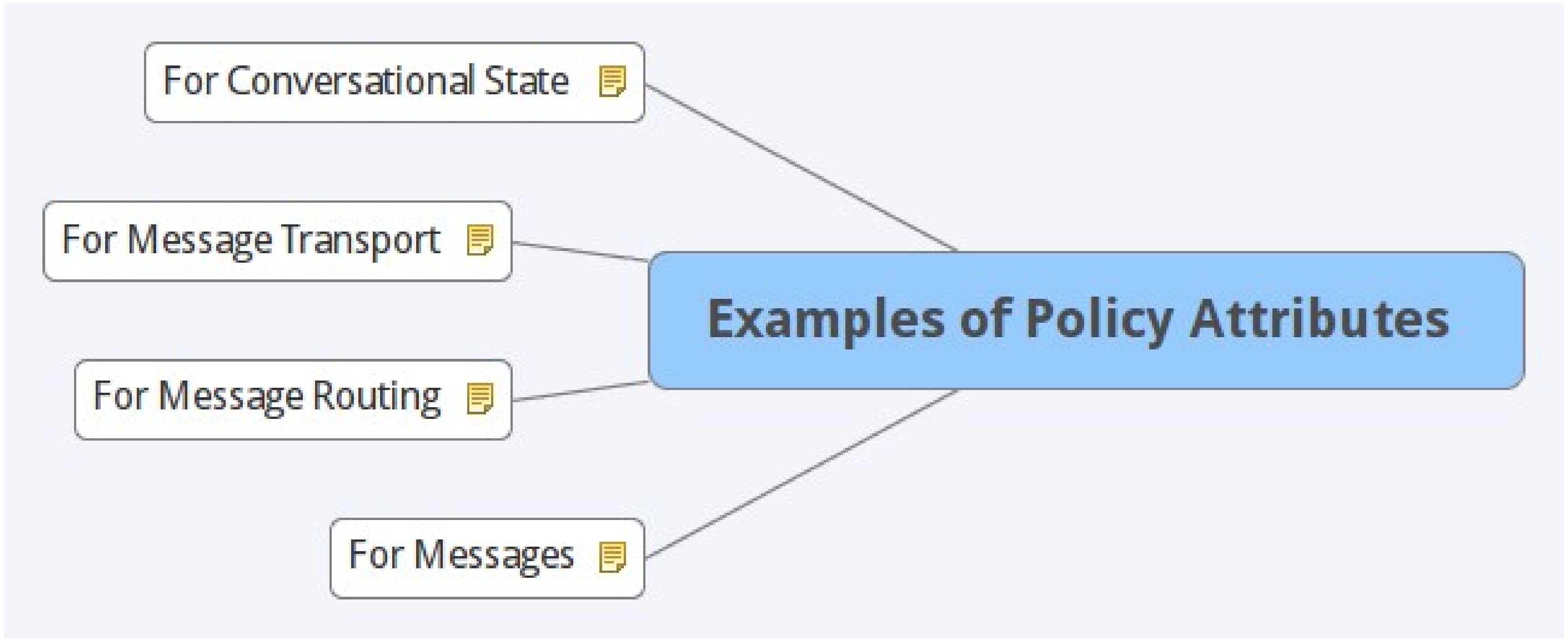
When operations are dynamically chained, based on the analysis of the contents of a message, it is a form of choreography.

As long as the RESPONSIBILITY of deciding about next steps is DEFERRED to the next component in the chain of events: it is a form orchestration. The conversation is decentralized.

Relevant standards: REST HATEOS, WS-CDL

(cc) BY-NC-SA

Examples of Policy Attributes



(cc) BY-NC-SA

For Messages

A policy defining, for example:

- Message exchanged as Document / Literal
- Message exchanged as Command
- Message containing an Metadata Header
- Message containing a trace of routes

(...)

(cc) BY-NC-SA

For Message Routing

A policy defining, for example:

- Routing rules
    - Based on a predefined logic sequence
    - Based on conversation state
    - Based on message contents
- Multi-cast / broadcast
    - Many service calls performed in parallel on different target component endpoints
    - Many messages broadcasted simultaneously on different transport channels

A policy defining, for example:

- One way
    - A simple notification
    - A simple invocation
- Two-way communication
    - Synchronous Request / response
    - Request / eventual response (asynchronous)

(cc) BY-NC-SA

For Message Transport

A policy defining, for example:

- Synchronous RPC / Message exchanges
- Asynchronous Message Exchanges

A policy defining, for example:

- Queue-based message communications
- Event-based message communications

A policy defining, for example:

- If the location of queues should be known and managed by calling components

(cc) BY-NC-SA

For Conversational State

A policy defining, for example:

- If the list of necessary conditions to commit a transaction are met
- If the list of necessary conditions to rollback a transaction are met

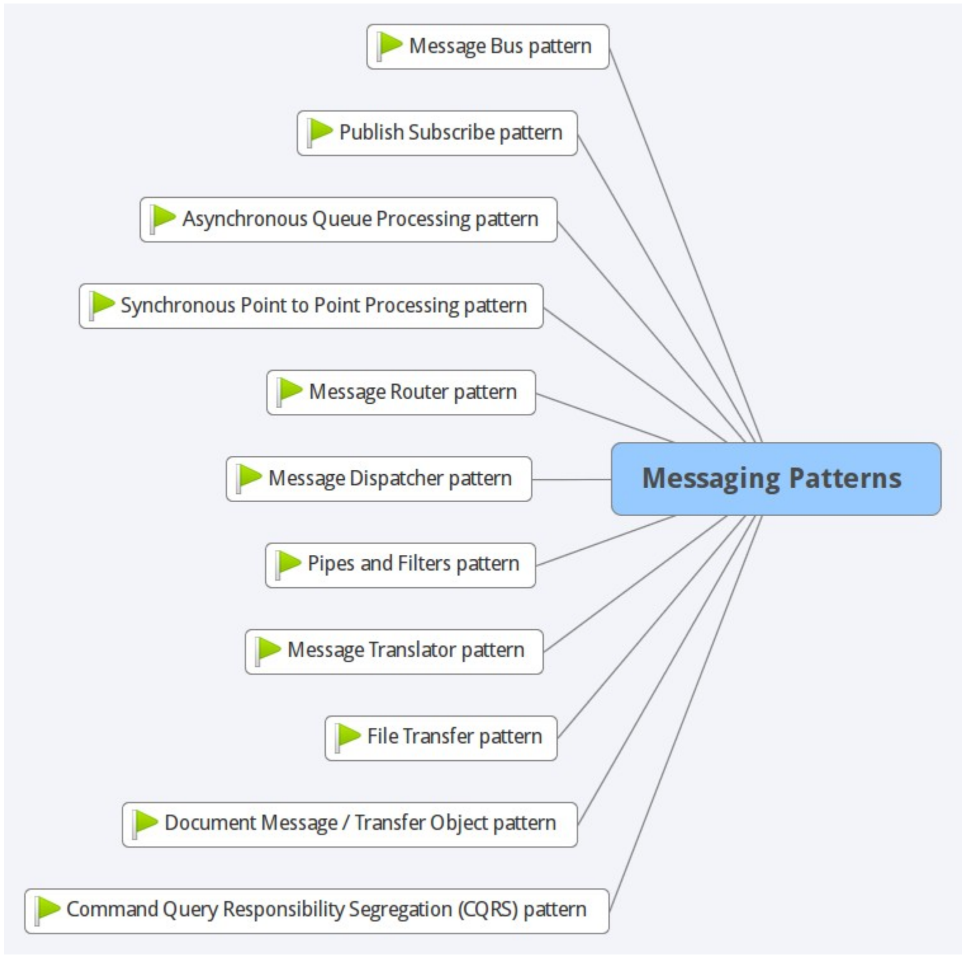
A policy defining, for example, the nature of the conditions themselves:

- Based on the list of steps previously successfully executed, what should be the next step?
- Acknowledgement of delivery within a given time-window
- Number of retries performed within a given time-window

(...)

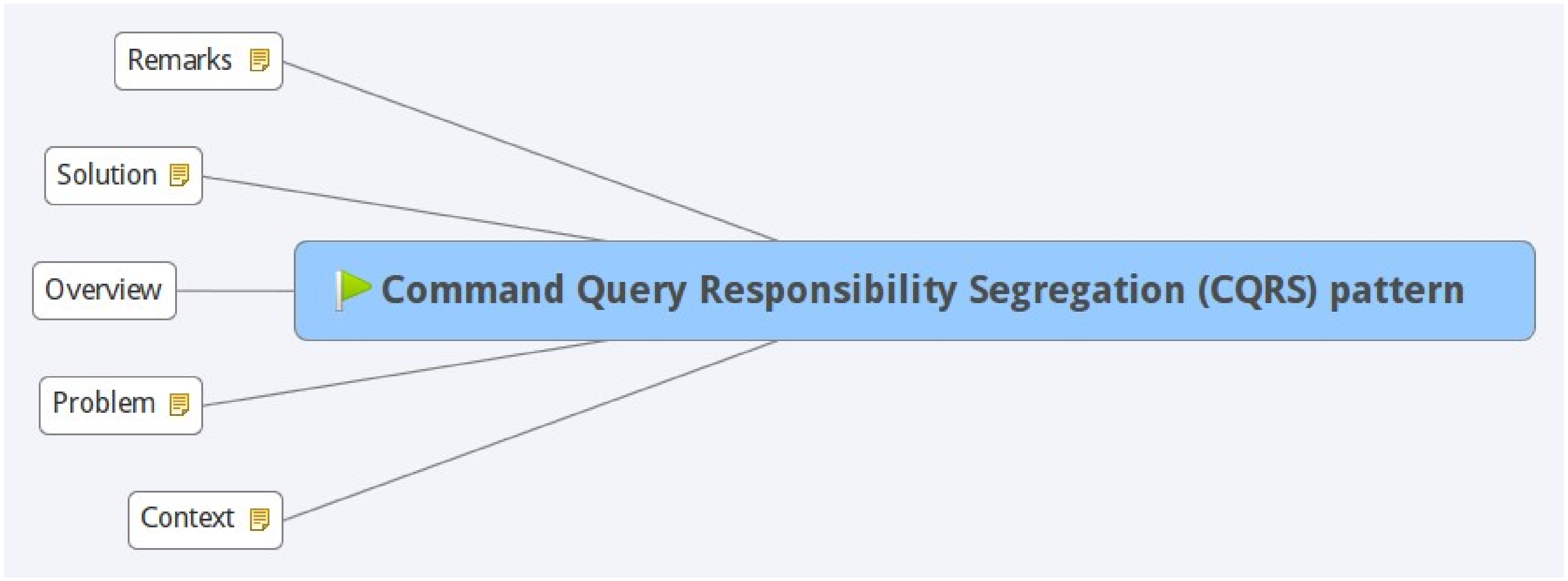
(cc) BY-NC-SA

Messaging Patterns


Message Bus pattern
Publish Subscribe pattern
Asynchronous Queue Processing pattern
Synchronous Point to Point Processing pattern
Message Router pattern
Message Dispatcher pattern
Messaging Patterns
Pipes and Filters pattern
Message Translator pattern
File Transfer pattern
Document Message / Transfer Object pattern
Command Query Responsibility Segregation (CQRS) pattern

BY-NC-SA

Command Query Responsibility Segregation (CQRS) pattern



cc BY-NC-SA

Context

Need to invoke functionality provided by other applications, not using Remote Procedure Invocation nut Messages, to meet fault-tolerance (reliability) requirements and interoperability/extensibility requirements.

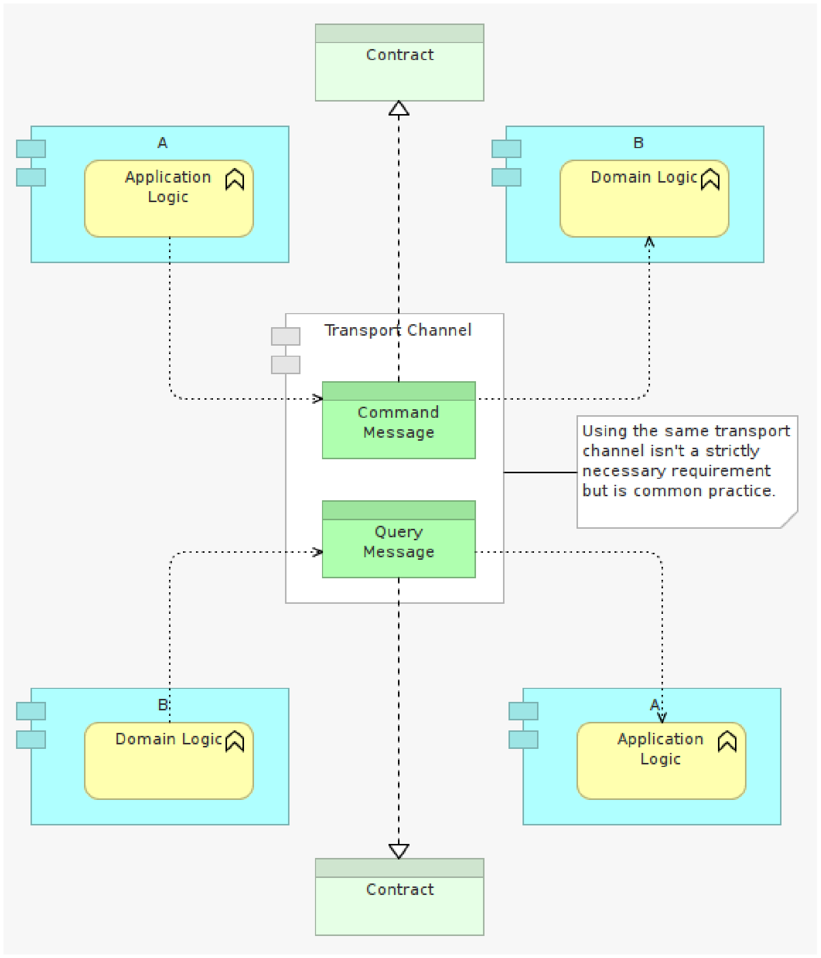
(cc) BY-NC-SA

Problem

How can a Message-Oriented architecture be used to invoke a procedure in another application?

cc BY-NC-SA

Overview



(cc) BY-NC-SA

Solution

Use a Command Message to reliably invoke an operation from another component.

This pattern states that every operation should either be a command that performs an action, or a query result that returns data to the caller, but not both.

Beyond these two distinction there is no specific message type for commands defined in the pattern; a Command Message is simply a regular message that happens to contain a command.

However, command–query separation is particularly well suited to a design by contract methodology, so to define specific types of commands and queries.

Command–query separation allows to retrace the commands exchanges between components overtime and as such is useful to determine the conversational state of an application.

(cc) BY-NC-SA

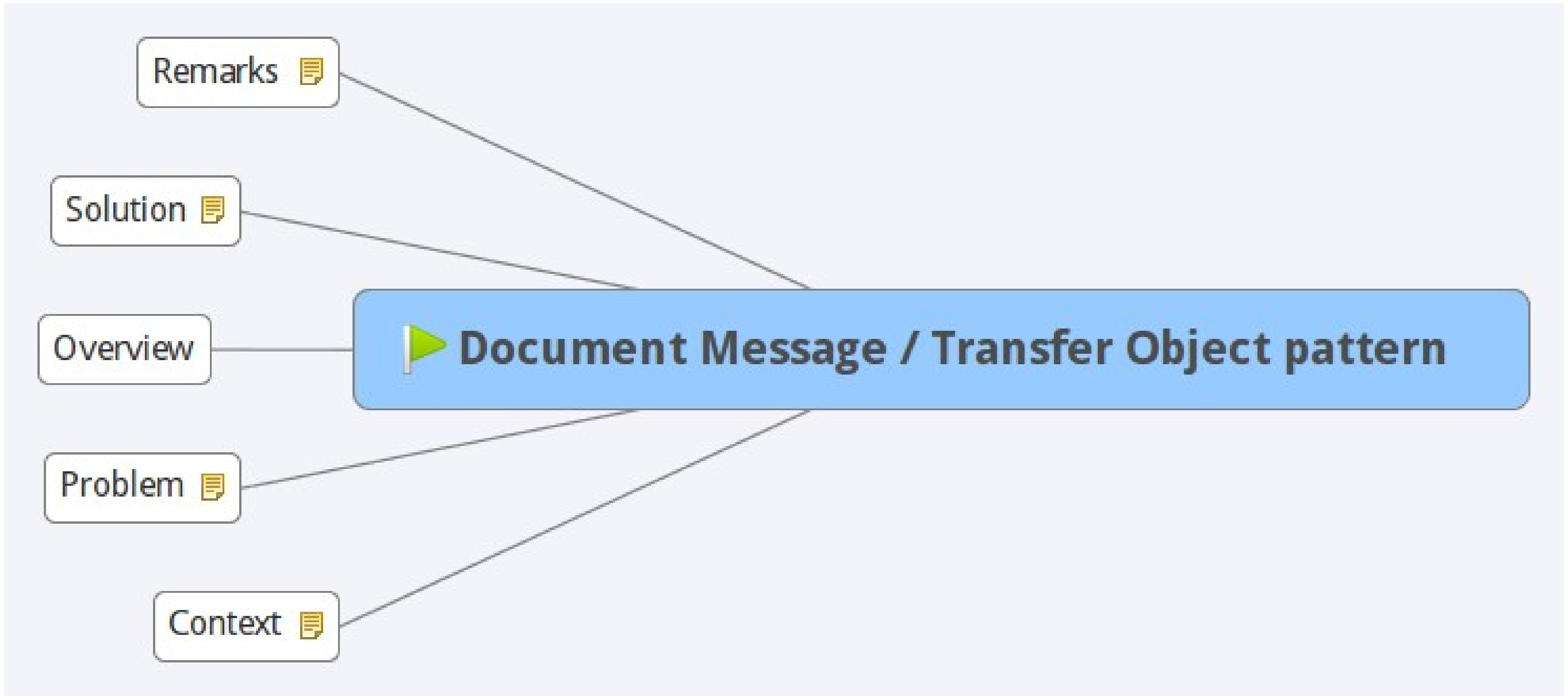
Remarks

The CQRS pattern has a simplifying effect on long-running, rich conversations between distributed components, making its states (via queries) and state changes (via commands) more comprehensible.

Implemented using queues the pattern makes great of use of reliable/guaranteed message delivery, improving the reliability of an architecture.

(cc) BY-NC-SA

Document Message / Transfer Object pattern



(cc) BY-NC-SA

Context

Application A required to perform a high number of RPC to Application B to execute its logic. All of these remote calls are costly, partly because of the amount of data marshalled/un-marshalled with each call.

Data currency (i.e. timeliness of the data exchanged is a concern) therefore ruling out file-transfer or shared database patterns.

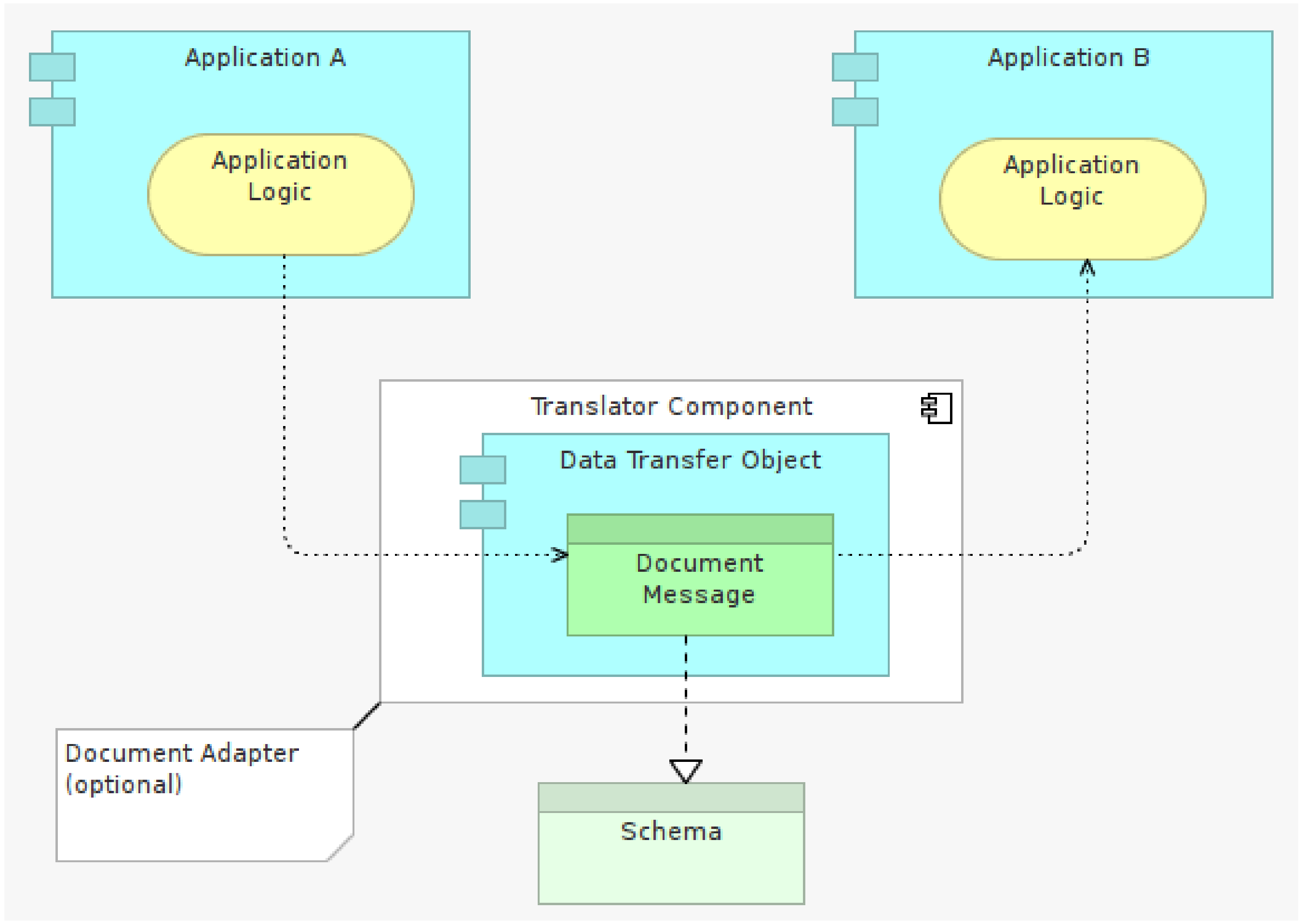
BY-NC-SA

Problem

How can messaging be used to optimized the conversation between 2 components? Is there a way to avoid having to serialize / de-serialize multiple argument calls?

(cc) BY-NC-SA

Overview



BY-NC-SA

Solution

Create a Document Message (also referred as Data Transfer Object) that can hold all the data pertaining to the RPC.

Use a synchronous Document/Literal Web Service, or an asynchronous Document Message channel to reliably transfer a data structure between application components.

cc BY-NC-SA

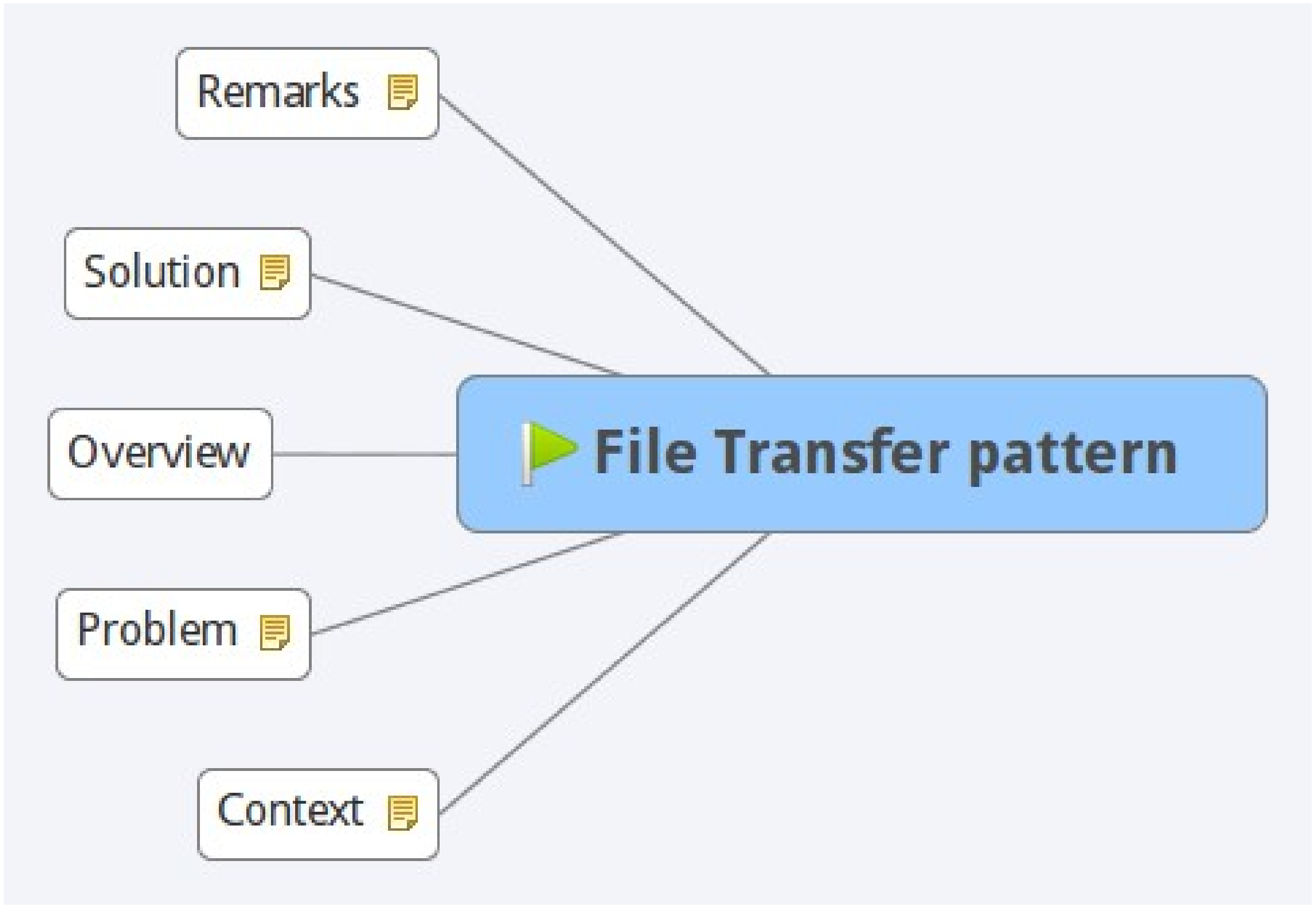
Remarks

The Document Message is a single unit of data, a single object or data structure conforming to a Schema definition.

Whereas a Command Message tells the receiver to invoke certain behavior, a Document Message just passes data and lets the receiver decide what, if anything, to do with the data.

BY-NC-SA

File Transfer pattern



(cc) BY-NC-SA

Context

An enterprise has multiple LEGACY applications that are being built independently, with different languages and platforms.

These application need to exchange information. These legacy applications do not have RPC capabilities (no interfaces endpoints to invoke).

(cc) BY-NC-SA

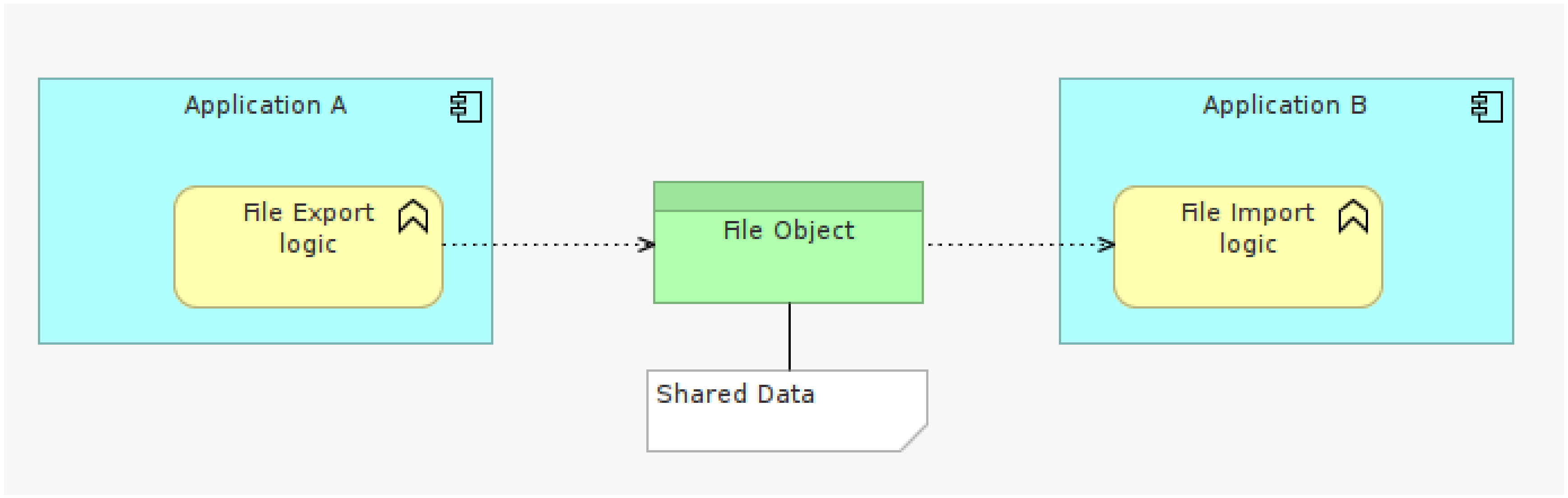
Problem

How do you integrate information systems that were not designed to work together?

How do you integrate multiple applications so that they work together and can exchange information?

(cc) BY-NC-SA

Overview



(cc) BY-NC-SA

Solution

Integrate applications at the logical data layer. Have each application produce structured (flat, XML or other) extract files containing information that other applications can to consume.

Make each application produce files that contain the information that the other applications must consume.

Produce the files at regular intervals according to the nature of the business. After a file is created, do not maintain the file.

BY-NC-SA

Remarks

An important decision with files is what format to use.

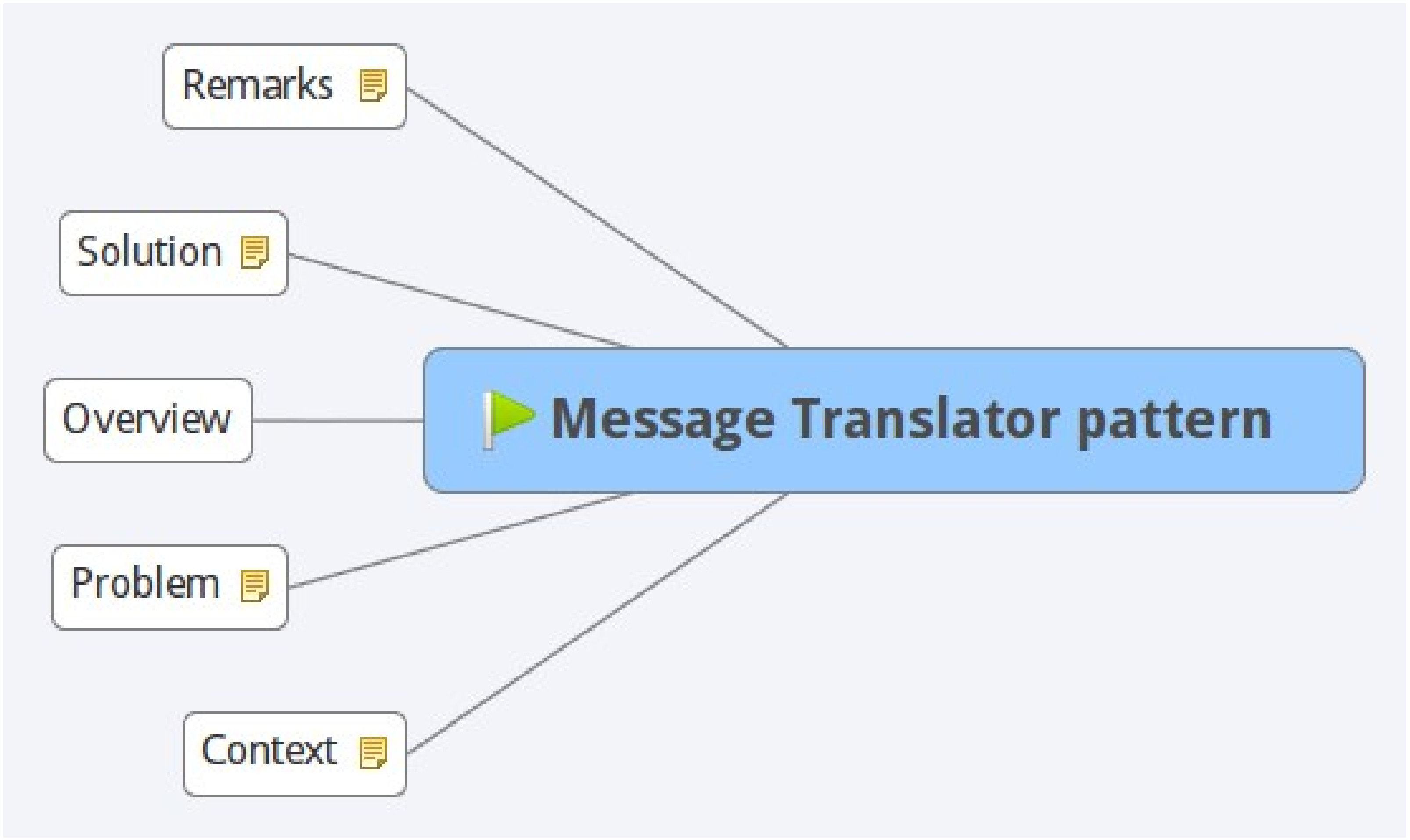
Very rarely will the output of one application be exactly what is needed for another.

Extract will require transformation down the line by downstream consumers.

Extract can follow a ISO (or other industry-specific) standard file format. Mainframe systems commonly extract VSAM files.

BY-NC-SA

Message Translator pattern



(cc) BY-NC-SA

Context

Need for a mechanism for converting a message payload from one representation to another as it flows through the messaging system.

Enterprise integration solutions route messages between existing applications such as legacy systems, packaged applications, homegrown custom applications, or applications operated by external partners.

Each of these applications is usually built around a proprietary data model. Each application may have a slightly different notion of the Customer entity.

The application's underlying data model usually drives the design of the physical database schema, an interface file format or a programming interface (API) -- those entities that an integration solution has to interface with. As a result, the applications expect to receive messages that mimic the application's internal data format.

Need to conform to data exchange standard body (ebXML, EDIFACT, ACORD).

(cc) BY-NC-SA

Problem

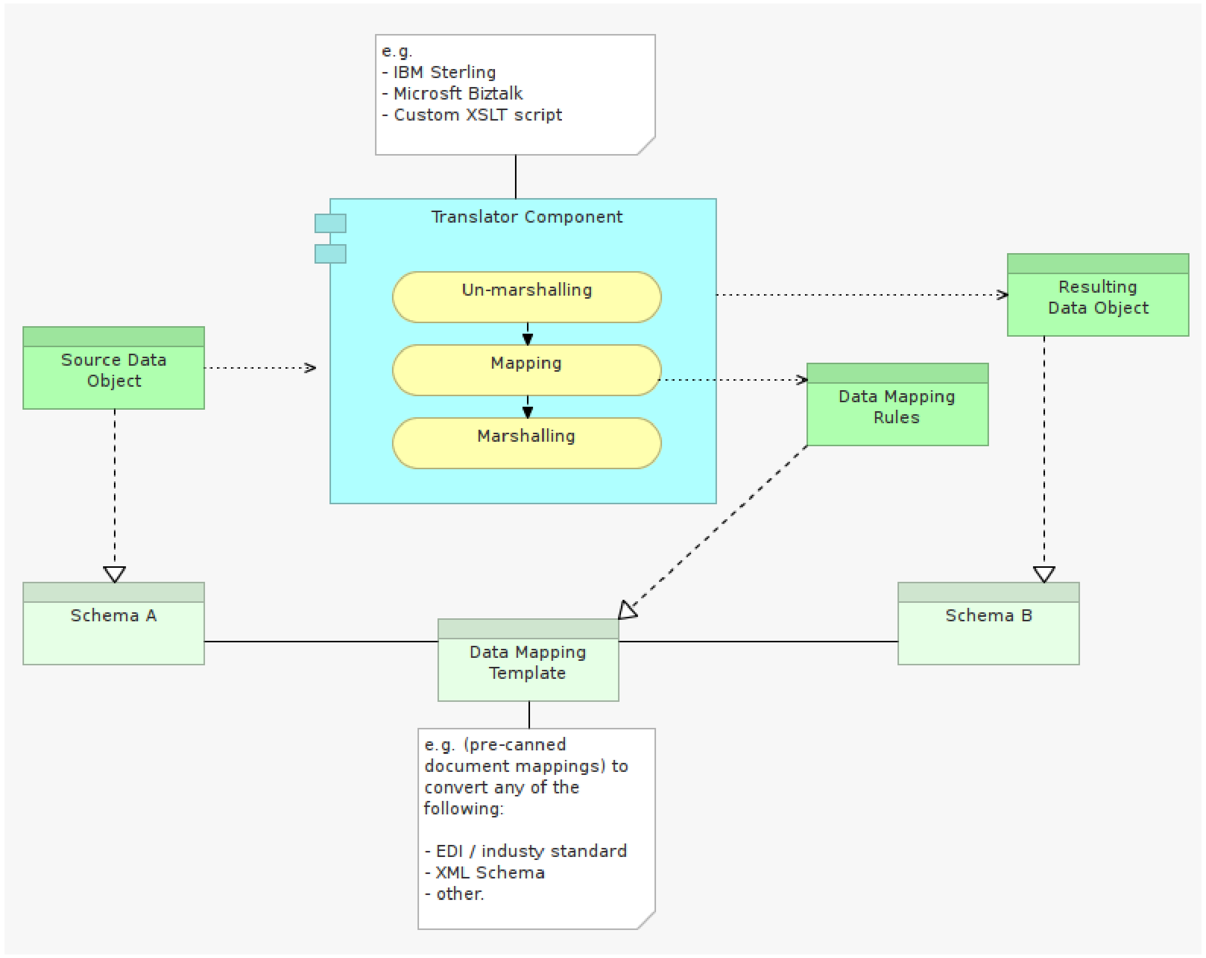
How can systems using different data formats communicate with each other using messaging?

Heterogeneous systems integration (legacy, in-house, and vendor provided)

may use different message representation for input or output.

(cc) BY-NC-SA

Overview



cc BY-NC-SA

Solution

Use a filter, referred as Message Translator to translate one data format into another, following data mapping specifications/rules.

Provide a system-independent mechanism for altering the message payload and metadata (envelope) prior to delivery to an application endpoint.

BY-NC-SA

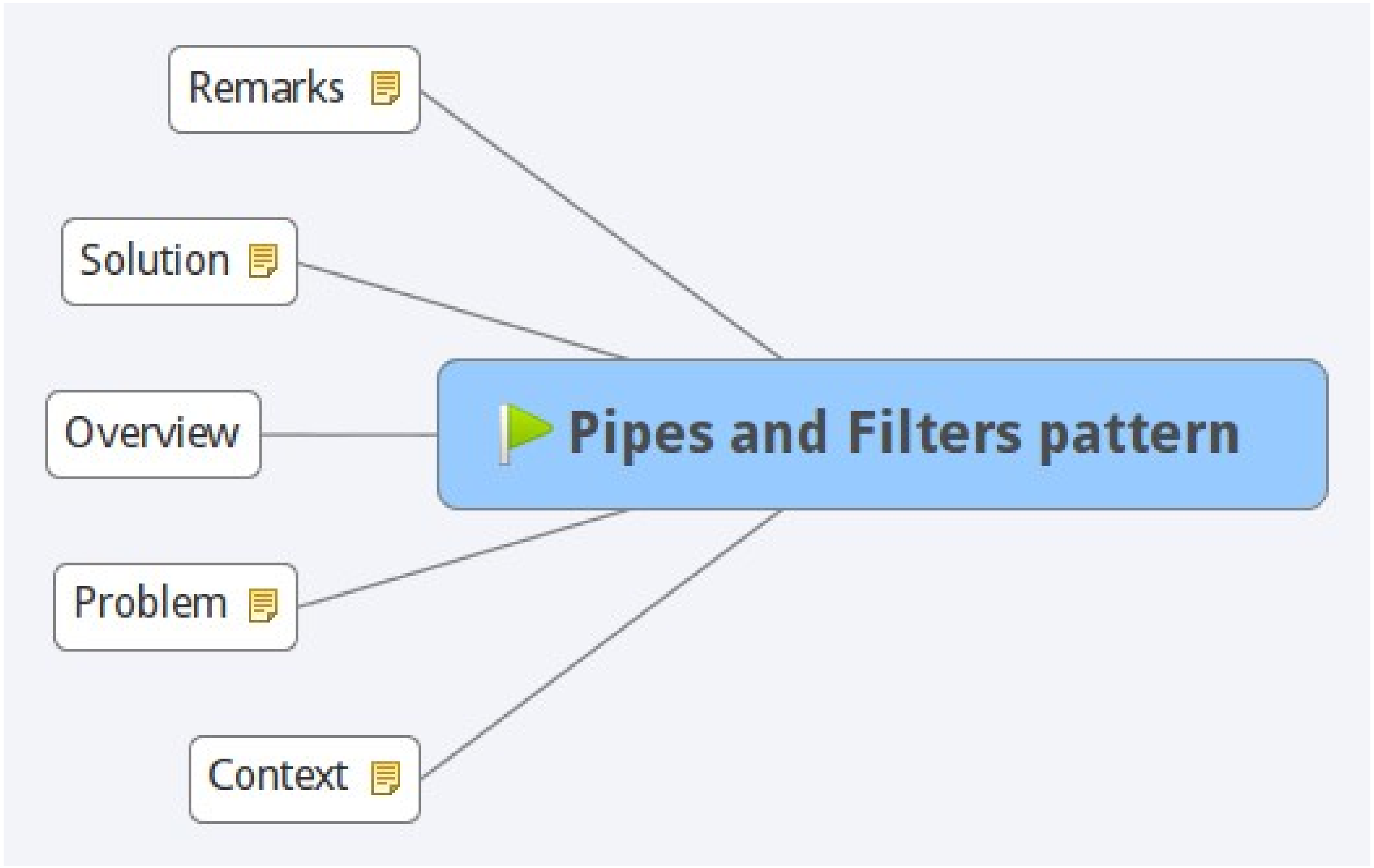
Remarks

The Message Translator is the messaging equivalent of the Adapter pattern described in GoF. An adapter converts the interface of a component into a another interface so it can be used in a different context.

Translators are one of the most effective message transformation mechanisms because they allow application developers and integrators to insulate, implement, test, and maintain these system components without modifying existing application workflow or domain/application logic.

BY-NC-SA

Pipes and Filters pattern



(cc) BY-NC-SA

Context

Many systems are required to transform streams of discrete data items, from input to output.

Need to implement flexible message processing between systems in a platform-independent manner and without introducing system dependencies or unnecessary coupling.

Need for a conduit that extracts data from a message, applies a transformation function to it as it flows between consumers and services.

Need for the integration solution to apply several transformations to the messages that are exchanged by its participants.

(cc) BY-NC-SA

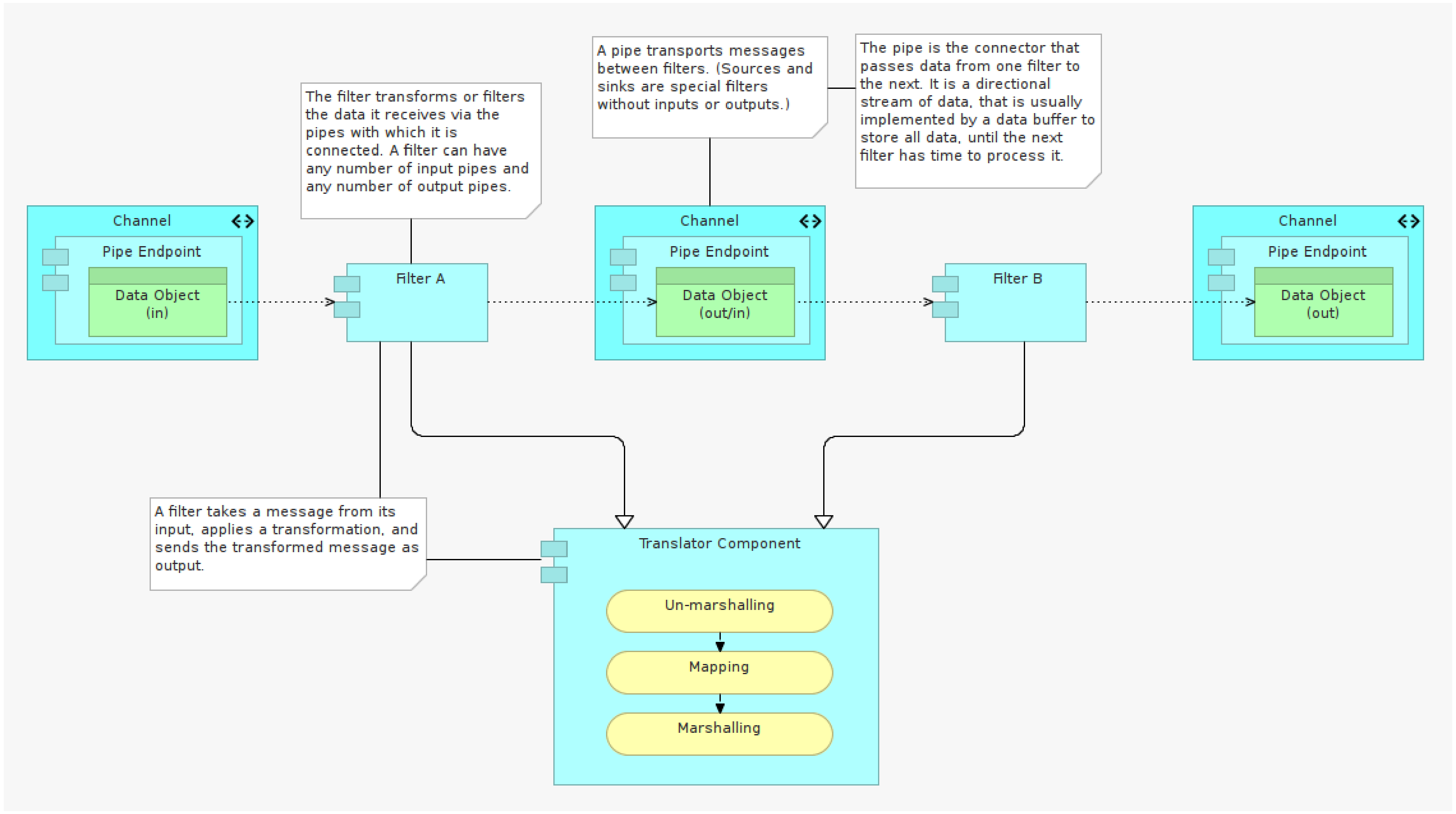
Problem

How do you implement a sequence of transformations so that you can combine and reuse them independently?

How to create a data transformation process generic and loosely coupled, so its parts can be re-used and executed in parallel, and be flexibly combined with each other?

(cc) BY-NC-SA

Overview


The filter transforms or filters the data it receives via the pipes with which it is connected. A filter can have any number of input pipes and any number of output pipes.

A pipe transports messages between filters. (Sources and sinks are special filters without inputs or outputs.)

The pipe is the connector that passes data from one filter to the next. It is a directional stream of data, that is usually implemented by a data buffer to store all data, until the next filter has time to process it.

A filter takes a message from its input, applies a transformation, and sends the transformed message as output.

cc BY-NC-SA

Solution

Implement the transformations by using a sequence of filter components, where each filter component receives an input message, applies a simple transformation, and sends the transformed message to the next component.

Conduct the messages through pipes that connect filter outputs and inputs and that buffer the communication between the filters.

The pattern of interaction in the pipe and filter pattern is characterized by successive transformations, of streams of data.

Data arrives at a filter input port, is transformed, and then passed via its output ports through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Filters order are interchangeable that enable different work flow functionality without changing the filters themselves.

(cc) BY-NC-SA

Remarks

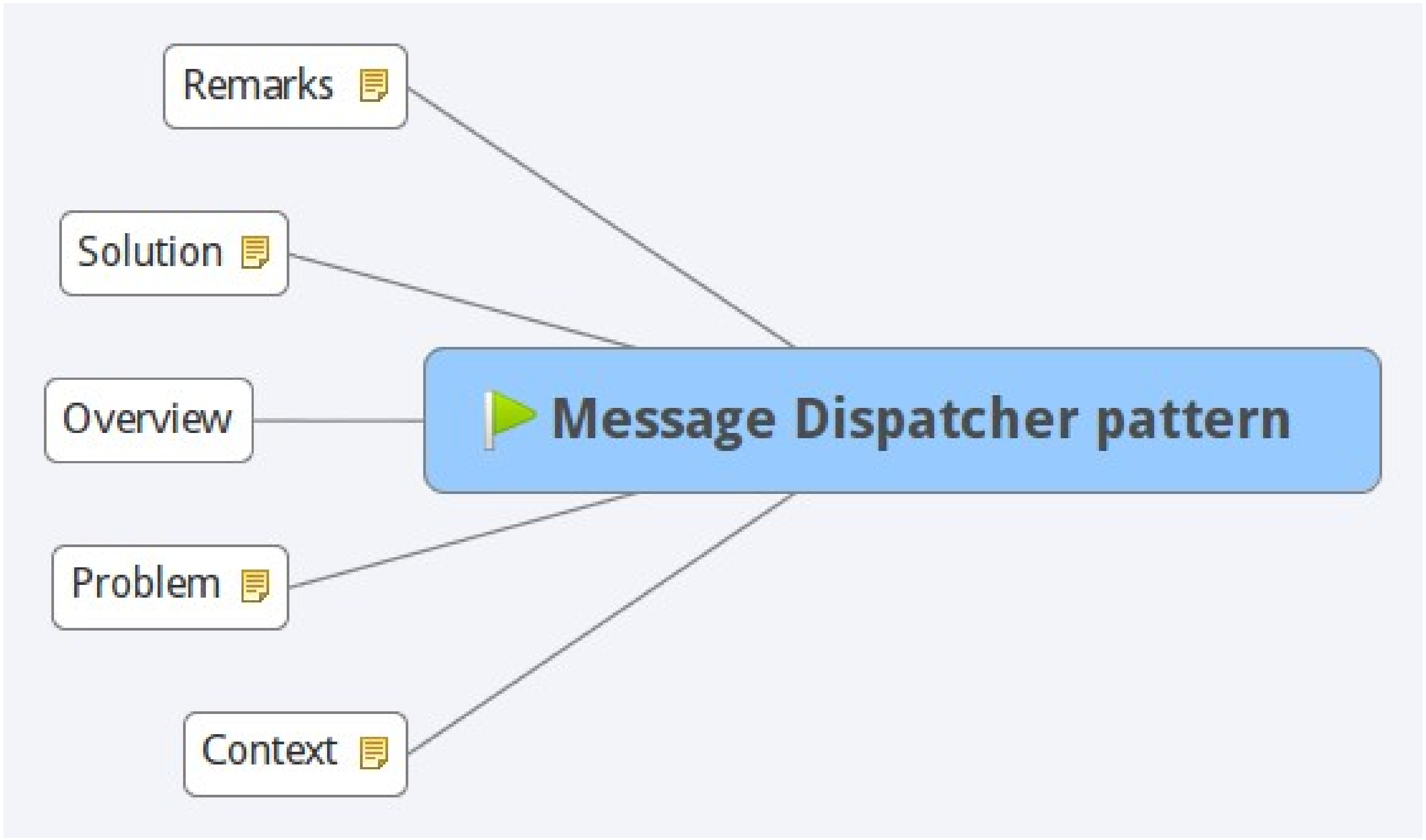
Must share the same external interface to facilitate integration and recombination. The point of the pipe & filters pattern eliminate data and dependencies by uniform defining a contract (inbound/outbound interface) that encourages reusability through composition.

Uses discrete functions on messages like encryption, data consolidation, redundancy elimination, data validation, etc.

Filters split larger processing tasks into discrete, easy to manage units that can be recombined for use by multiple service providers.

(cc) BY-NC-SA

Message Dispatcher pattern



(cc) BY-NC-SA

Context

Need for a general mechanism for dispatching a number of messages to one or more destinations, in no particular order, based on configurable rules or filters applied to message payloads.

(cc) BY-NC-SA

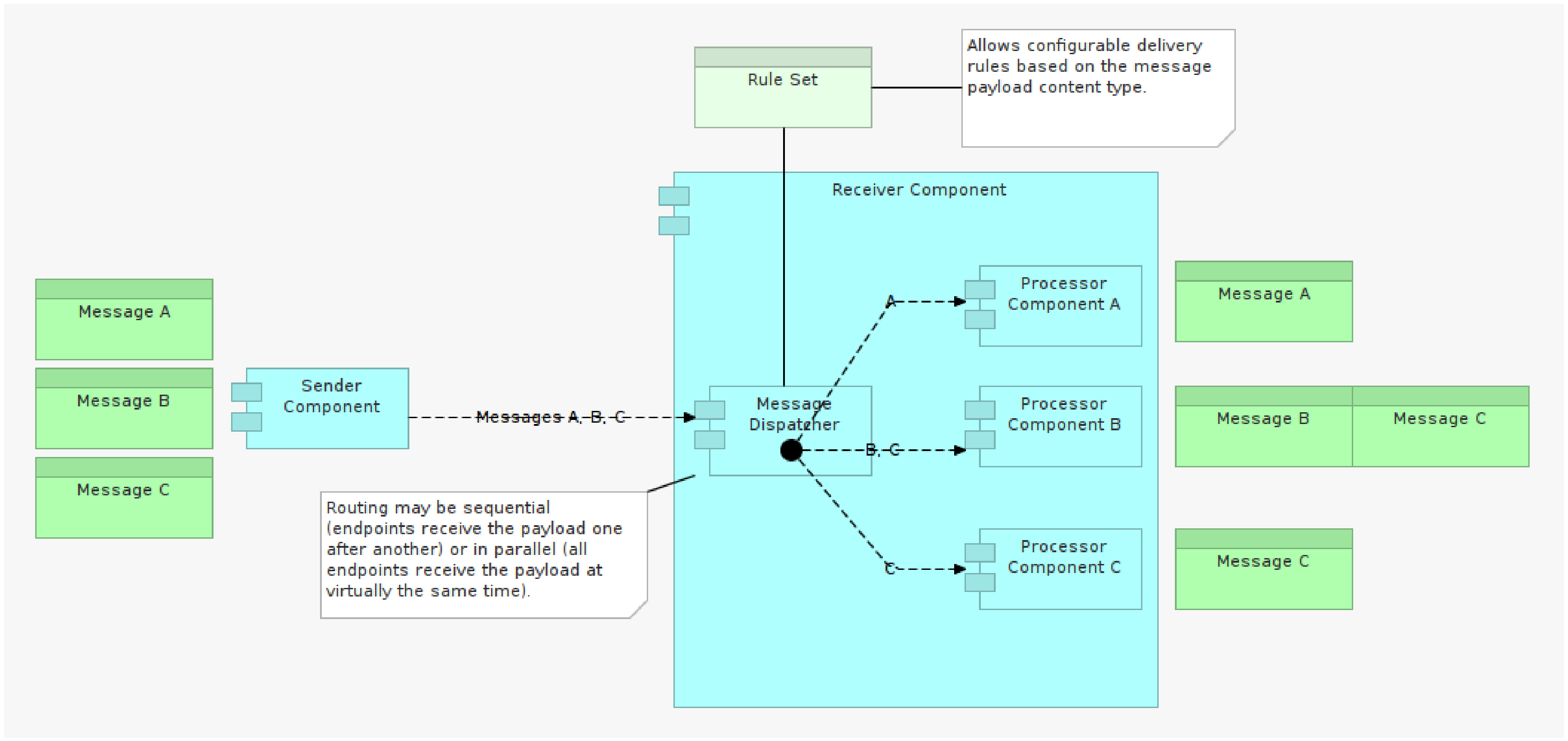
Problem

How can multiple consumers on a single channel coordinate their message processing?

An application must connect with one or more application endpoints without coupling itself with any of them.

cc BY-NC-SA

Overview


Rule Set
Allows configurable delivery rules based on the message payload content type.
Receiver Component
Message A
Message B
Message C
Sender Component
Messages A, B, C
Message Dispatcher
A
B, C
C
Processor Component A
Processor Component B
Processor Component C
Routing may be sequential (endpoints receive the payload one after another) or in parallel (all endpoints receive the payload at virtually the same time).
Message A
Message B
Message C
Message C

(cc) BY-NC-SA

Solution

Create a Message Dispatcher on a channel that will consume messages from a channel and distribute them to performers.

Use a conduit that allows configurable delivery rules based on the message payload, data filters, or content type.

Routing may be sequential (endpoints receive the payload one after another) or in parallel (all endpoints receive the payload at virtually the same time).

cc BY-NC-SA

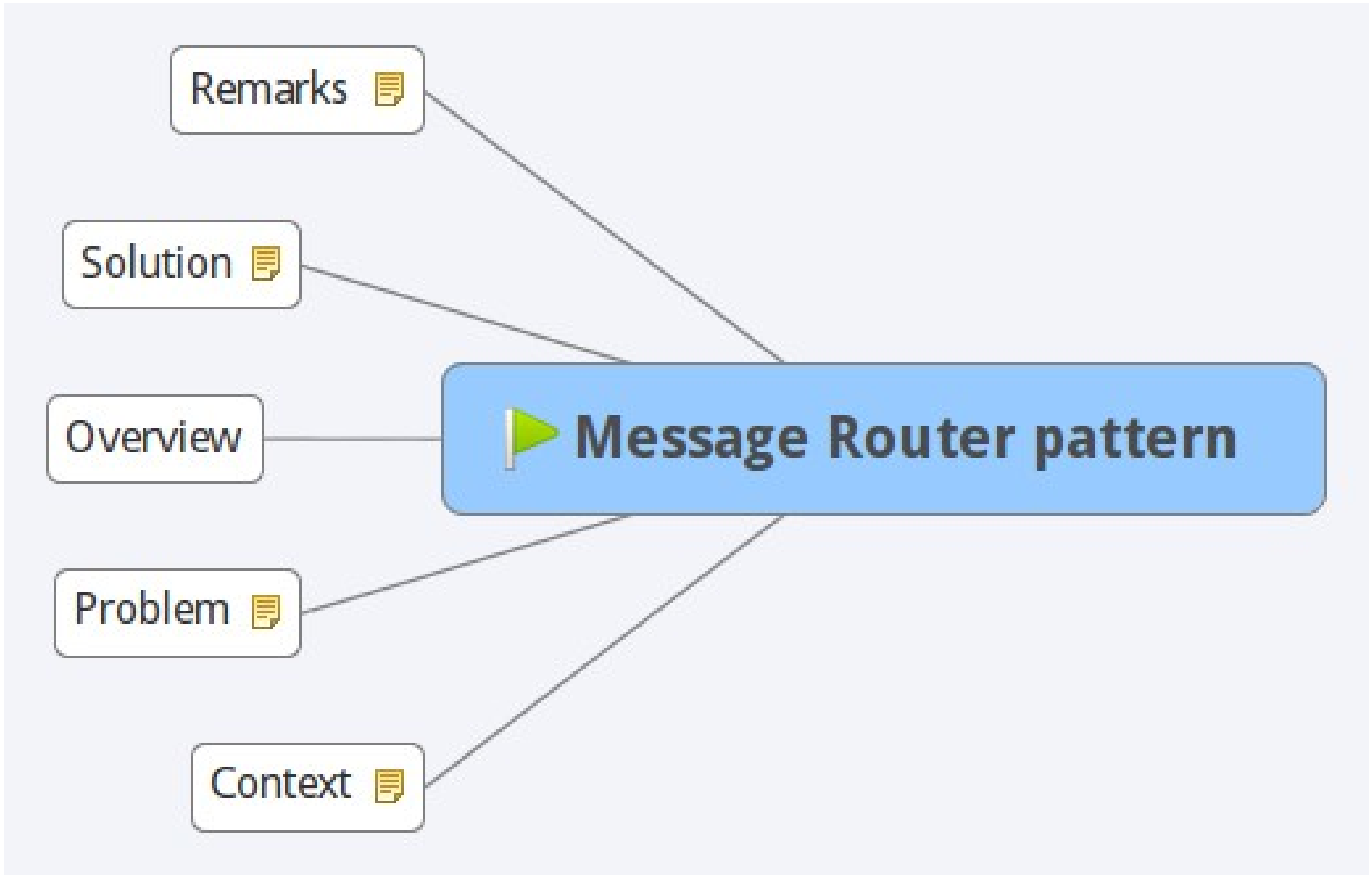
Remarks

The router abstraction is in use in all modern SOA systems in some forms, whether available in queuing or bus-based systems out of the box, or implemented in custom-made applications and message delivery systems because they provide an elegant and simple mechanism for system independent

message delivery.

(cc) BY-NC-SA

Message Router pattern



(cc) BY-NC-SA

Context

Routing messages through a distributed system based on filtering rules

is inefficient because messages are sent to every destination’s filter and

router for inspection and rules resolution, whether the message could be

processed or not.

Need for an efficient mechanism for dispatching messages to one or more

destinations based on configurable, non-filtering rules applied to

the message payload.

(cc) BY-NC-SA

Problem

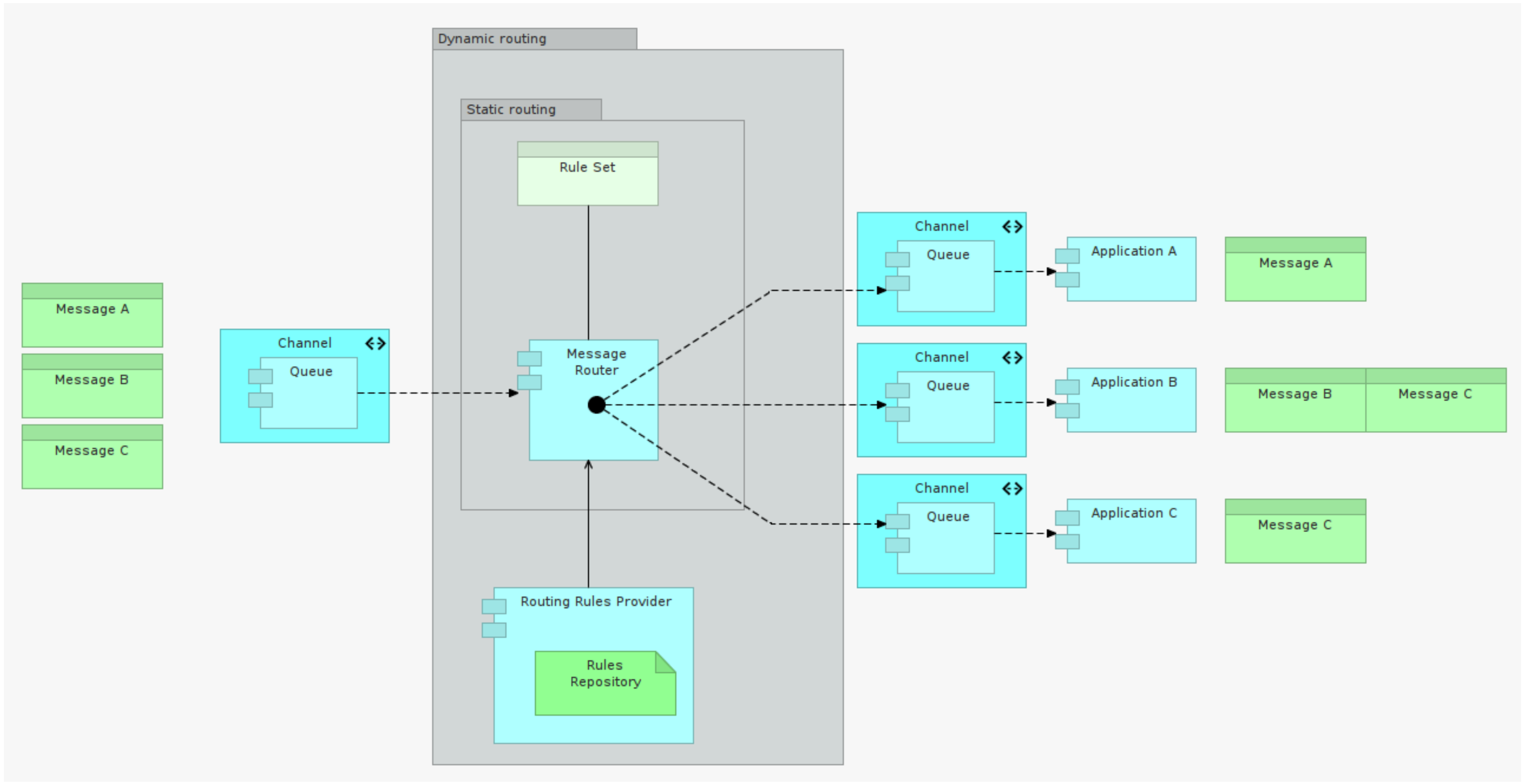
How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?

Message dispatching based on application-specific data elements such as customer attributes, message type, etc.

You need to know the network address of the other side.

BY-NC-SA

Overview



CC BY-NC-SA

Solution

Define a message router that includes both filtering rules and knowledge about the processing destination paths so that messages are delivered only to the processing endpoints that can act upon them.

Unlike filters, message routers do not modify the message content and are only concerned with message destination.

Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel channel depending on a set of conditions.

(cc) BY-NC-SA

Remarks

Better overall message delivery and processing performance at the cost of increased delivery system complexity since the router must implement both knowledge of the destinations and heuristic, arbitrary rules.

This pattern is excellent for decoupling applications by removing routing information from discrete systems.

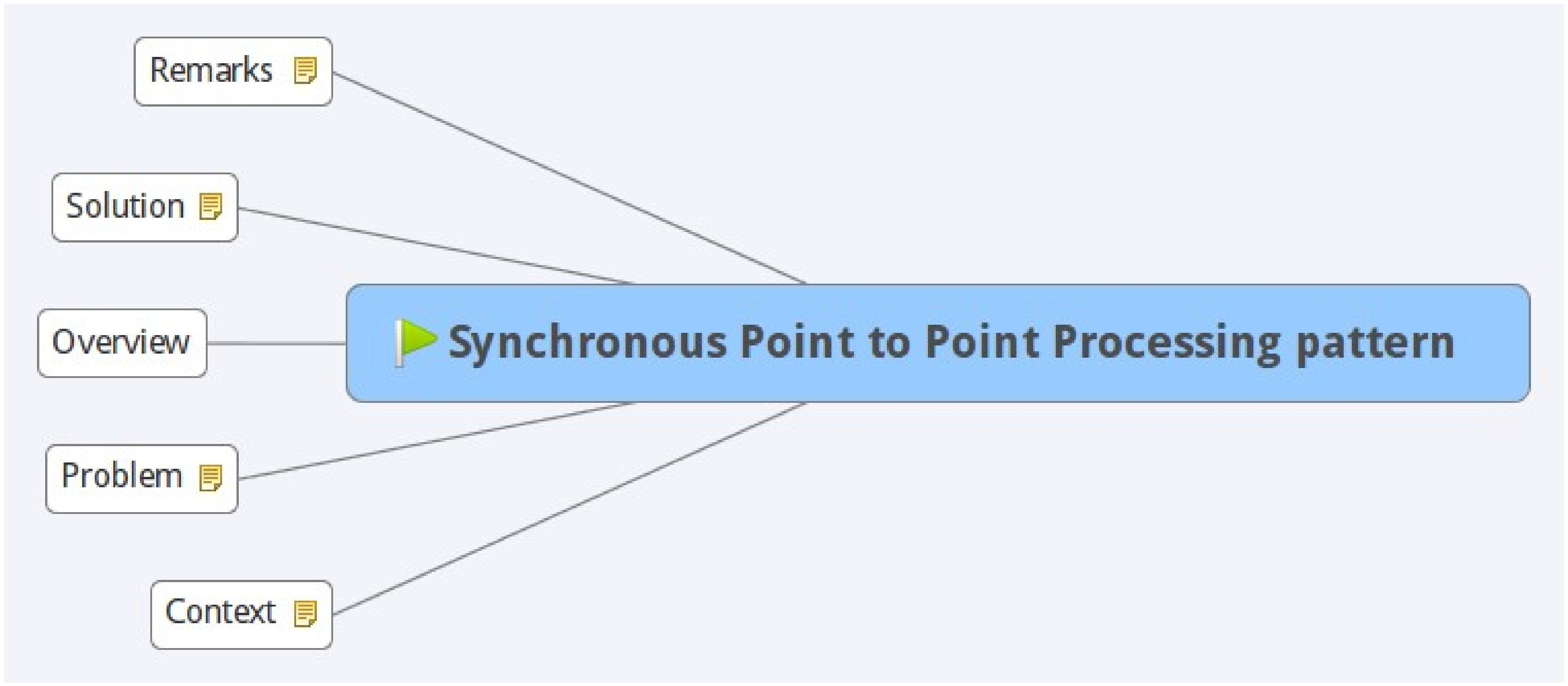
The Message Router differs from the most basic notion of Pipes and Filters in that it connects to multiple output channels.

Thanks to the Pipes and Filters architecture the components surrounding the Message Router are completely unaware of the existence of a Message Router.

This pattern adopts the mindset that all messages are distributed, and thus subject to failure on the network. In the future era of cloud computing, that’s probably going to end up being true for all solutions.

(cc) BY-NC-SA

Synchronous Point to Point Processing pattern



(cc) BY-NC-SA

Context

Need for a variable amount of distributed components of a same type to consume messages originating from a same producer, with a view to maximize performance.

BY-NC-SA

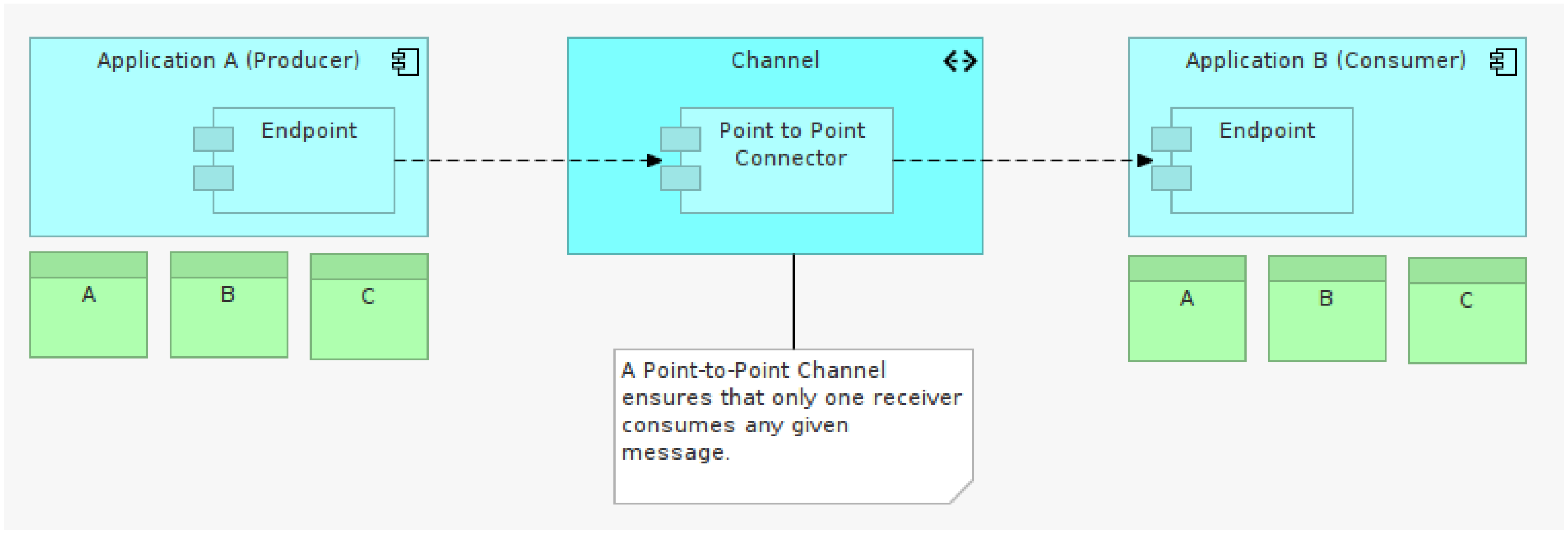
Problem

How to maximize the use of distributed components processing messages in parallel?

How can the caller be sure that exactly one receiver will receive the document or perform the call?

cc BY-NC-SA

Overview



(cc) BY-NC-SA

Solution

Send the message on a Point-to-Point Channel, which ensures that only one receiver will receive a particular message.

A Point-to-Point Channel ensures that only one receiver consumes any given message.

If the channel has multiple receivers, only one of them can successfully consume a particular message.

cc BY-NC-SA

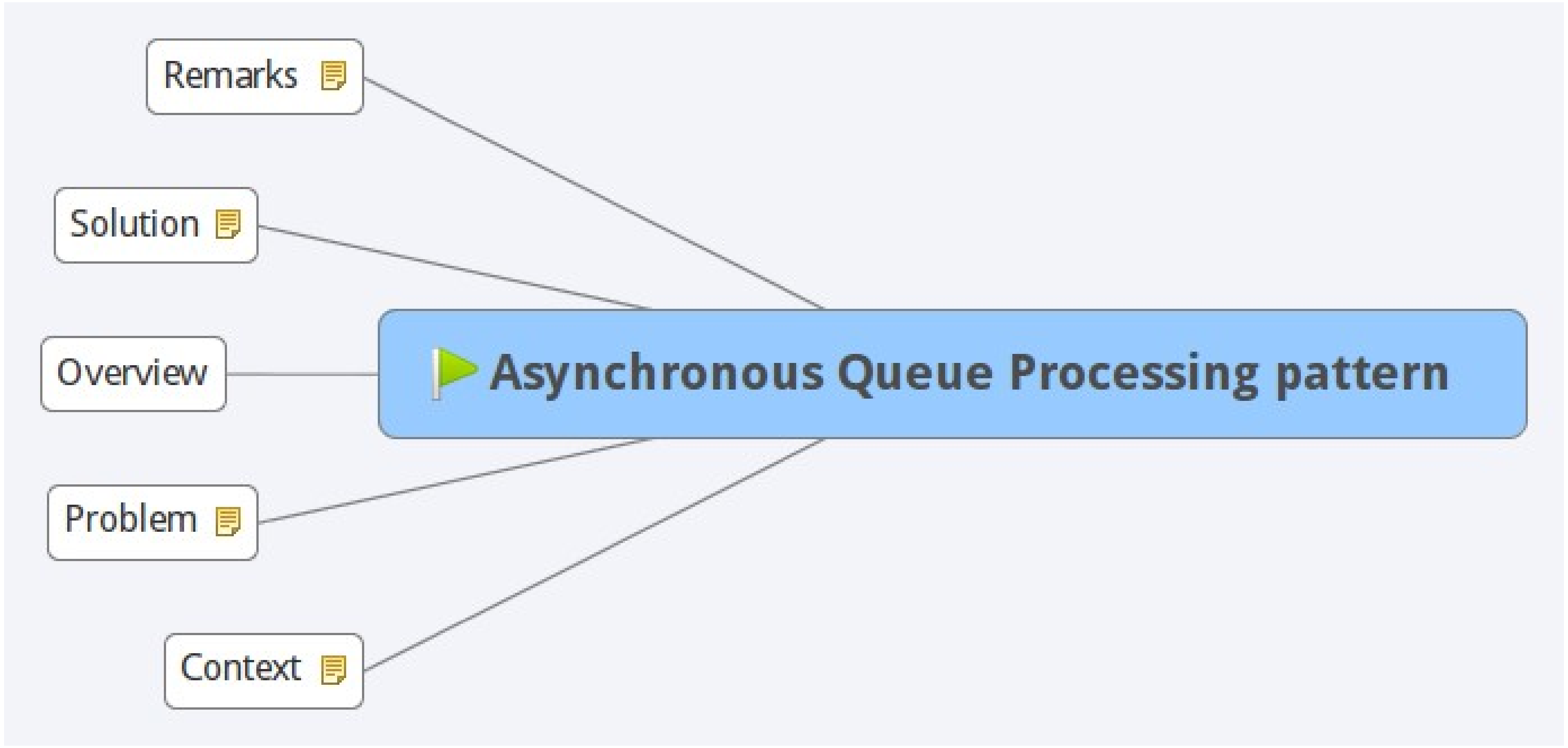
Remarks

If multiple receivers try to consume a single message, the channel ensures that only one of them succeeds, so the receivers do not have to coordinate with each other.

The channel can still have multiple receivers to consume multiple messages concurrently, but only a single receiver consumes any one message.

cc BY-NC-SA

Asynchronous Queue Processing pattern



cc BY-NC-SA

Context

Need for a mechanism for queuing messages between one or more distributed component endpoints to decouple processing time and resources for each stage of a processing work flow.

(cc) BY-NC-SA

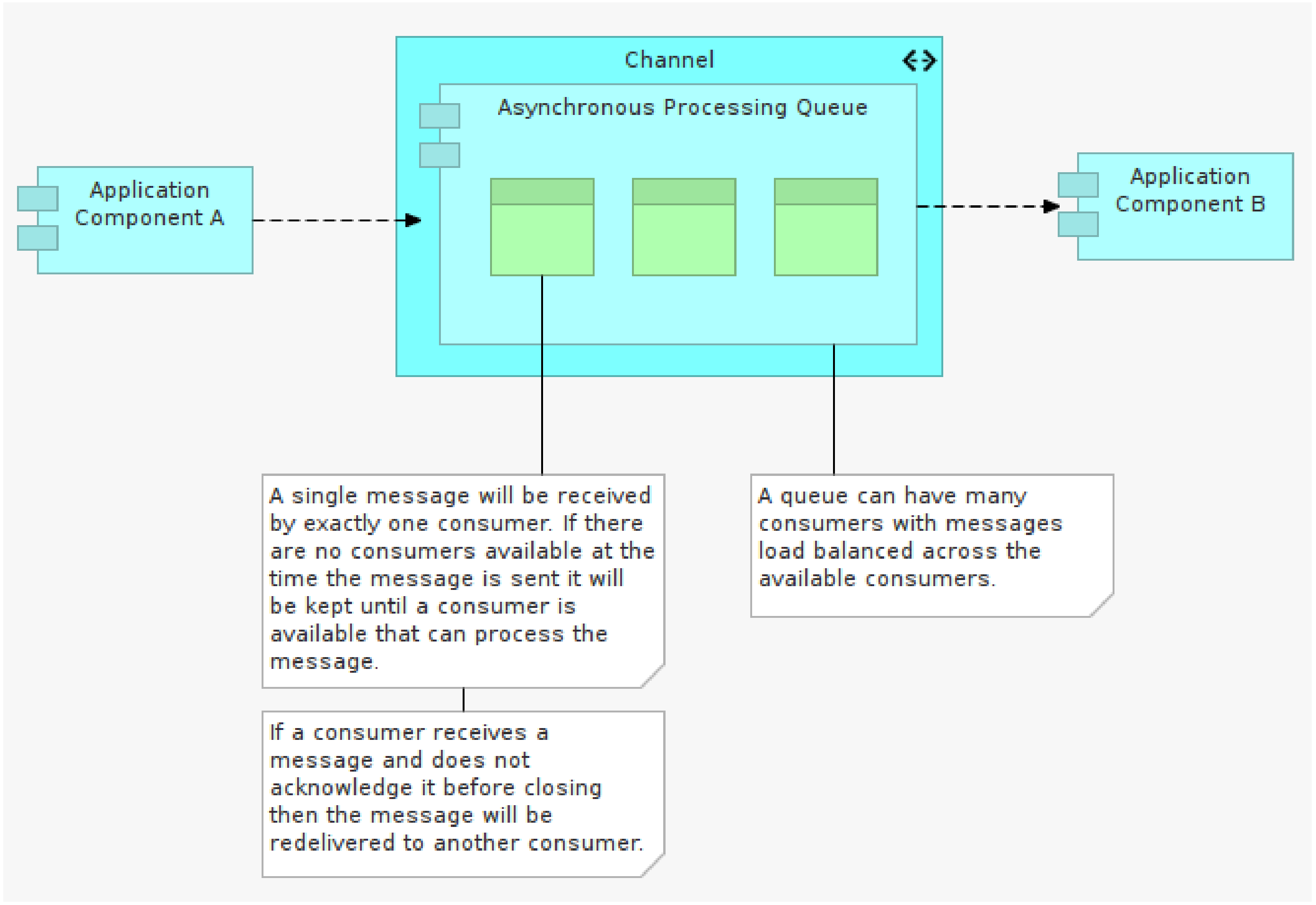
Problem

How to increase reliability of messages exchanged between distributed components?

How to guarantee that messages have been delivered so to certify that the exchanges meet the conversation policy specified the by architecture?

BY-NC-SA

Overview



A single message will be received by exactly one consumer. If there are no consumers available at the time the message is sent it will be kept until a consumer is available that can process the message.

A queue can have many consumers with messages load balanced across the available consumers.

If a consumer receives a message and does not acknowledge it before closing then the message will be redelivered to another consumer.

BY-NC-SA

Solution

Consumers exchange messages with the services through a processing queue that decouples front-end (message capture) from the back-end (processing); messages arrive into the queue at a rate different from that of processing.

Sending a message does not require both systems to be up and ready at the same time.

A single message will be received by exactly one consumer. If there are no consumers available at the time the message is sent it will be kept until a consumer is available that can process the message.

If a consumer receives a message and does not acknowledge it before closing then the message will be redelivered to another consumer.

(cc) BY-NC-SA

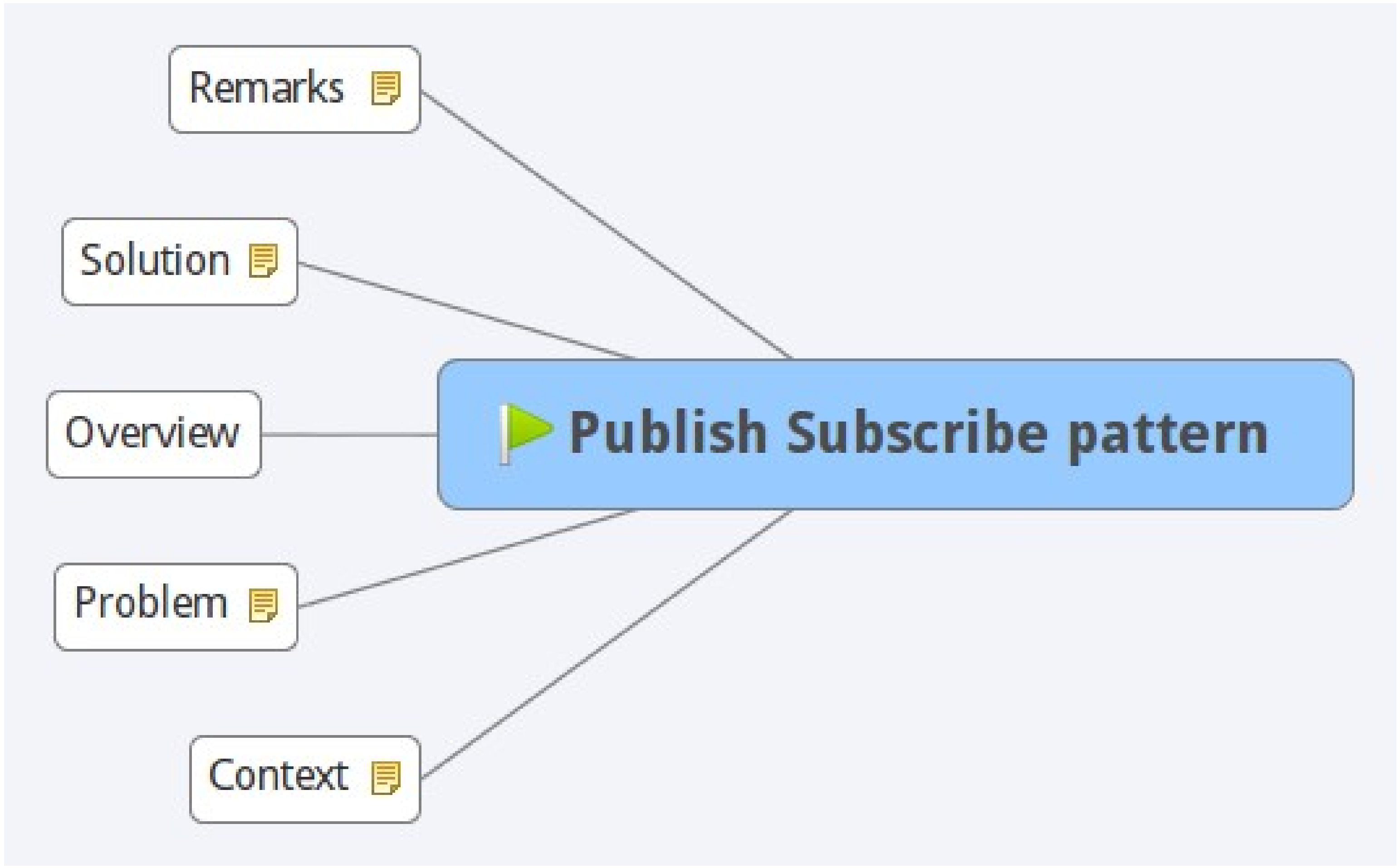
Remarks

The Asynchronous Processing Queue Processing pattern suited for applications that require scalability of the front-end (e-commerce) and back-end (such as mainframe data consolidation).

Processing queues are well-understood and scale horizontally or vertically, depending on the application requirements.

A number of solid open-source and commercial and several reference implementations exist, based on standardized API.

Publish Subscribe pattern



(cc) BY-NC-SA

Context

Need for a mechanism allowing routing messages to consumers in response to specific events or triggers.

Consumers must process messages as they become available in a system.

(cc) BY-NC-SA

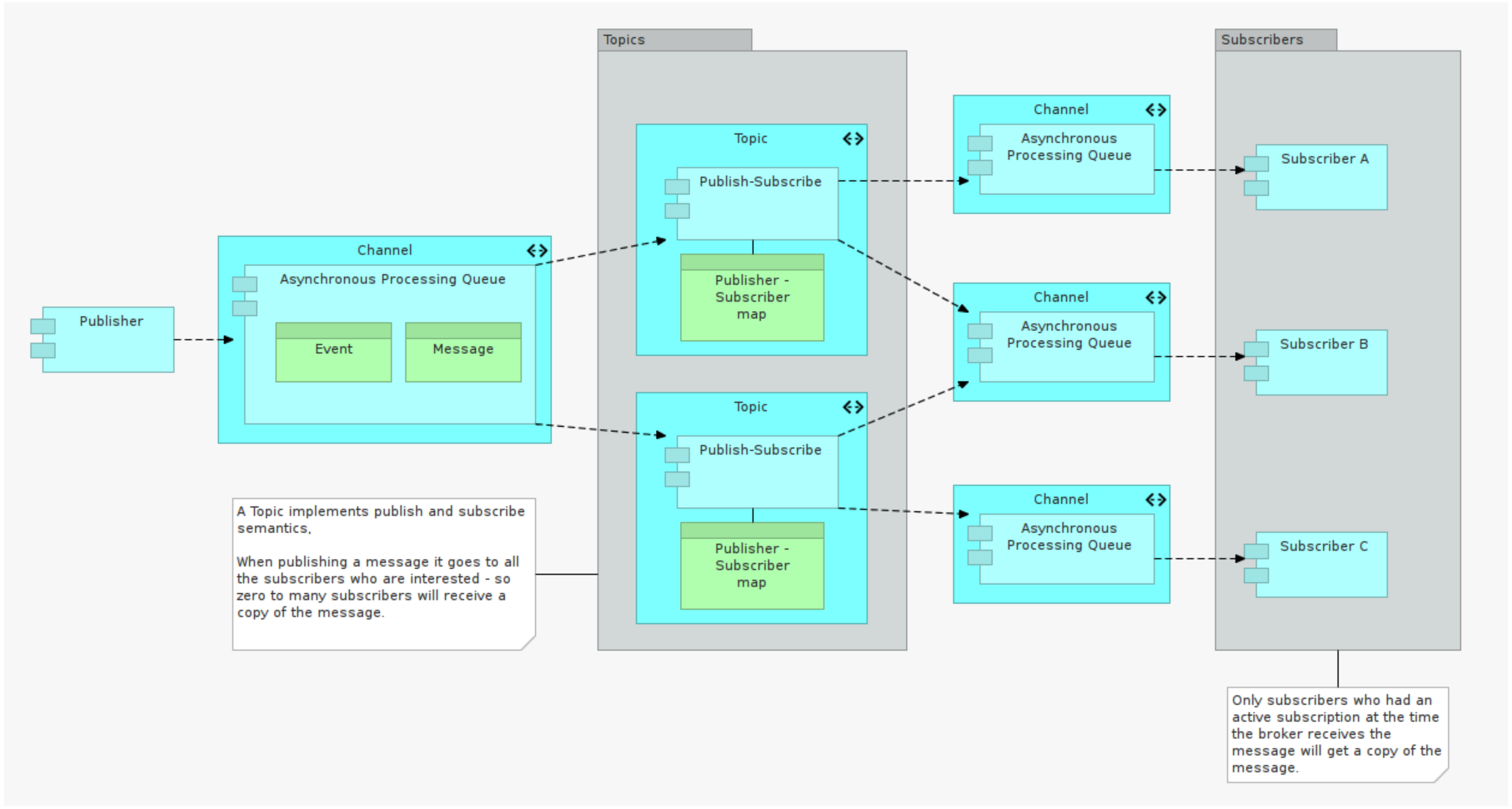
Problem

How can a sender component broadcast an event or message to a finite and configurable set interested receiver components?

How can an application in an integration architecture only send messages to the applications that are interested in receiving the messages without knowing the identities of the receivers?

(cc) BY-NC-SA

Overview


Topics
Subscribers
Publisher
Channel
Asynchronous Processing Queue
Event
Message
Topic
Publish-Subscribe
Publisher - Subscriber map
Topic
Publish-Subscribe
Publisher - Subscriber map
Channel
Asynchronous Processing Queue
Channel
Asynchronous Processing Queue
Channel
Asynchronous Processing Queue
Subscriber A
Subscriber B
Subscriber C

A Topic implements publish and subscribe semantics,

When publishing a message it goes to all the subscribers who are interested - so zero to many subscribers will receive a copy of the message.

Only subscribers who had an active subscription at the time the broker receives the message will get a copy of the message.

(cc) BY-NC-SA

Solution

Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each subscribed receiver.

A Publish-Subscribe channel has one input channel that splits into multiple output channels, one for each subscriber.

When an event is published into the channel, the Publish-Subscribe Channel delivers a copy of the message to each of the output channels.

Each output channel has only one subscriber, which is only allowed to consume a message once. In this way, each subscriber only gets the message once and consumed copies disappear from their channels.

Enable listening applications to subscribe to specific messages.

(cc) BY-NC-SA

Remarks

There are three variations of Topics for creating a mechanism that sends messages to all interested subscribers.

- List-Based Publish/Subscribe

- Broadcast-Based Publish/Subscribe

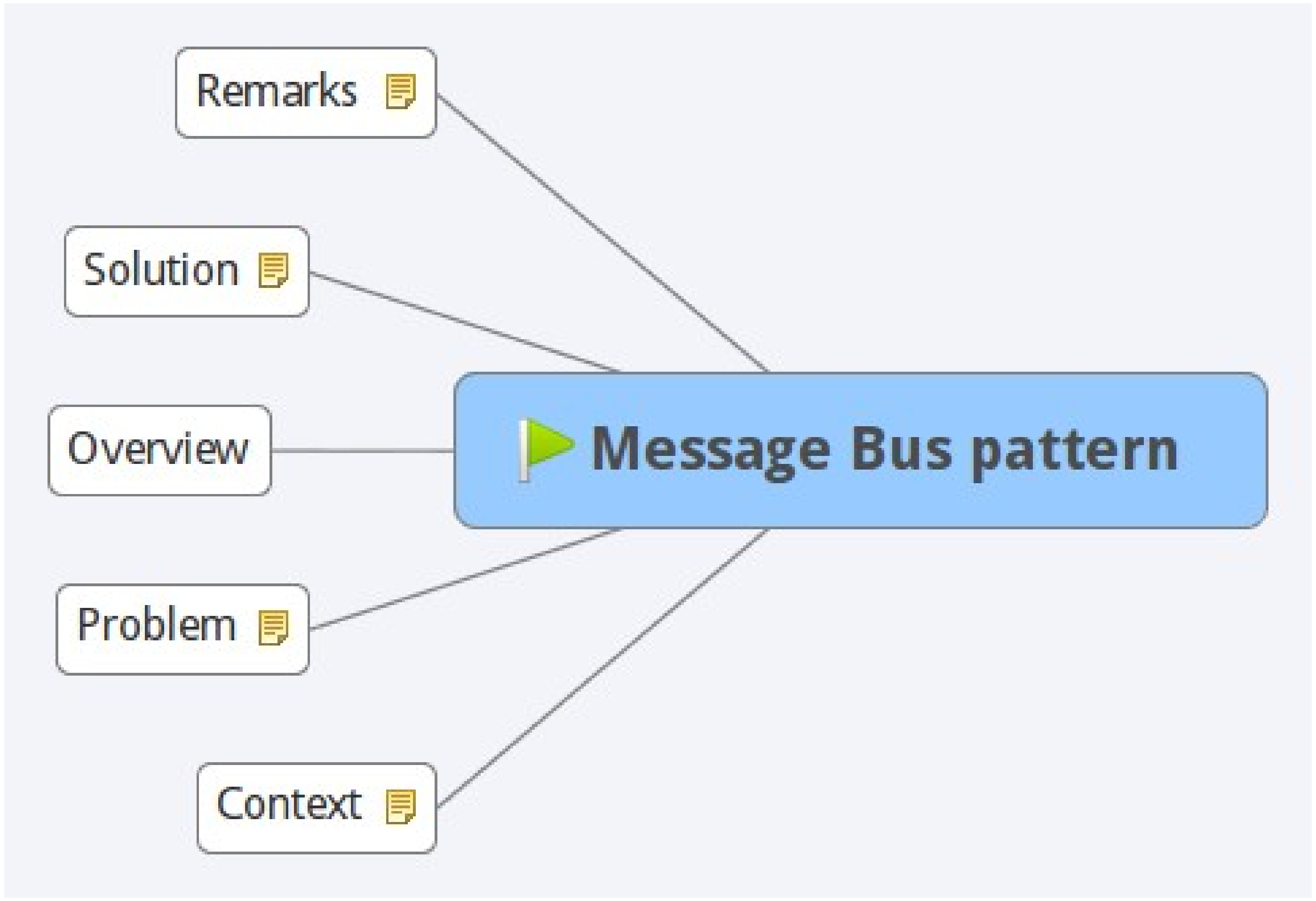
- and Content-Based Publish/Subscribe

The conversation policy of an architecture can be refactored by dynamically inserting new Topics.

Examples: JMS, Kafka

(cc) BY-NC-SA

Message Bus pattern



(cc) BY-NC-SA

Context

Need to integrate applications that are provided by different vendors and communicate via different communication protocols /proprietary formants.

These applications run on a variety of platforms. Some of these applications generate messages and many other applications consume the messages.

Architecture requires required integration of heterogeneous systems legacy and new system interoperability, protocol abstraction.

(cc) BY-NC-SA

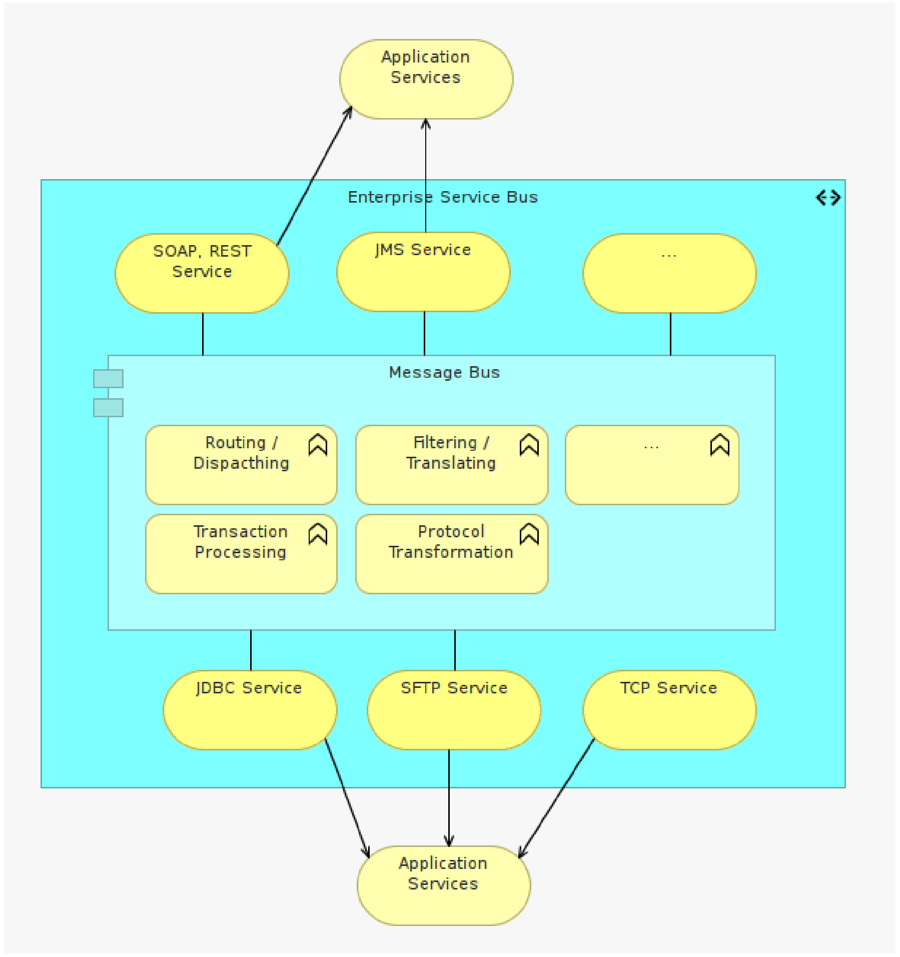
Problem

How can I integrate multiple applications so that they work together and can exchange information?

As an integration solution grows, how can you lower the cost of rationalizing (modernizing, adding or removing) applications?

(cc) BY-NC-SA

Overview



BY-NC-SA

Solution

Connect all applications through a logical component known as a message bus. A Service Bus specializes in transporting messages between applications.

A Service Bus uses Messaging to transfer packets of data frequently, immediately, reliably, and synchronously, using customizable formats.

A bus contains three key elements: a set of agreed-upon message schemas; a set of common command messages, and a shared infrastructure for sending bus messages to recipients.

It provide a data- or protocol-neutral conduit with abstract entry and exit points for interconnecting applications independently of their underlying technology.

(cc) BY-NC-SA

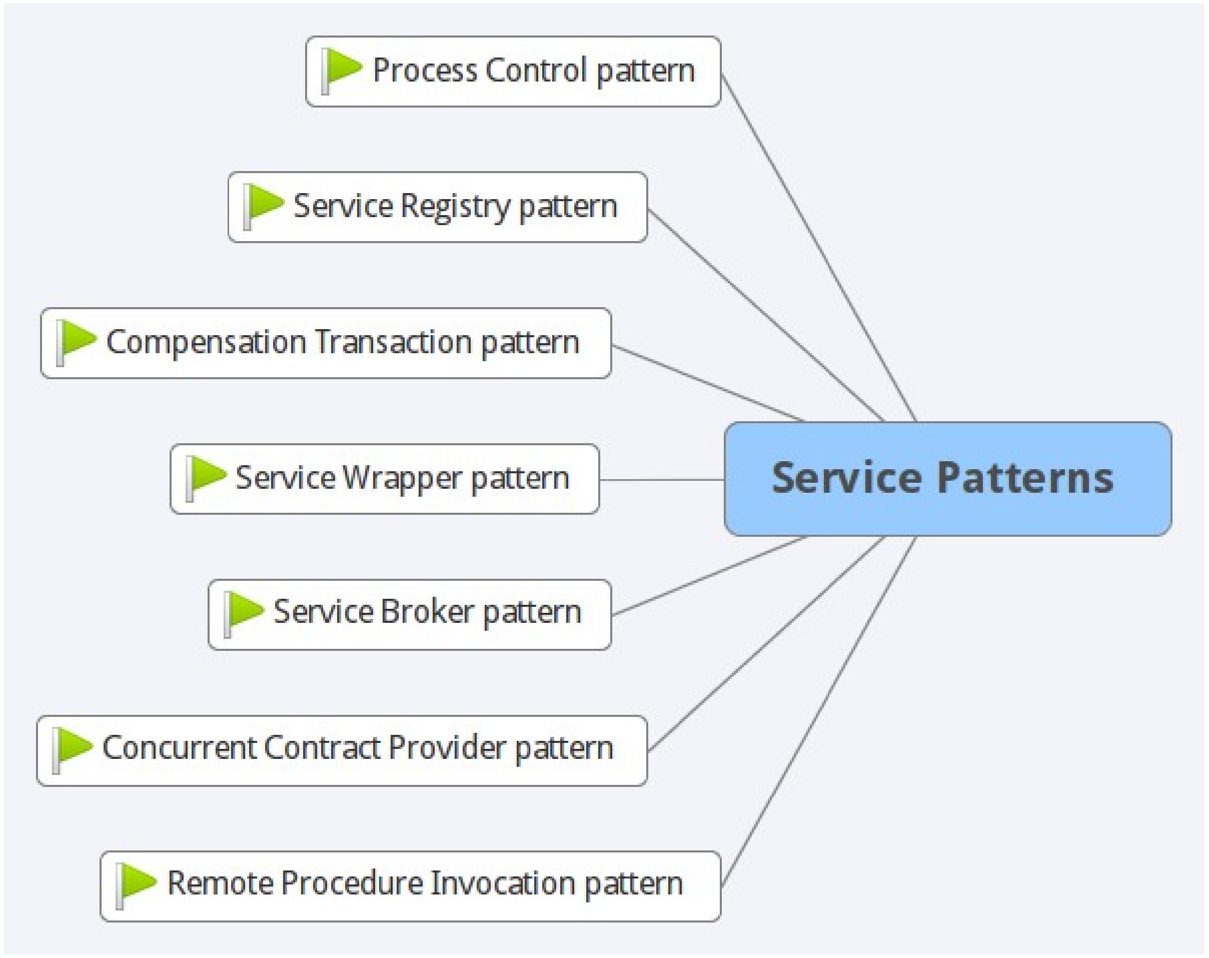
Remarks

Thinking about the communication in an asynchronous manner invite architects to recognize that working with a remote application is slower, which encourages design of components with high cohesion (lots of work locally) and low adhesion (selective work remotely).

Message Bus platforms consolidate all Messaging patterns. These platforms is a pragmatic reaction to the problems of distributed systems.

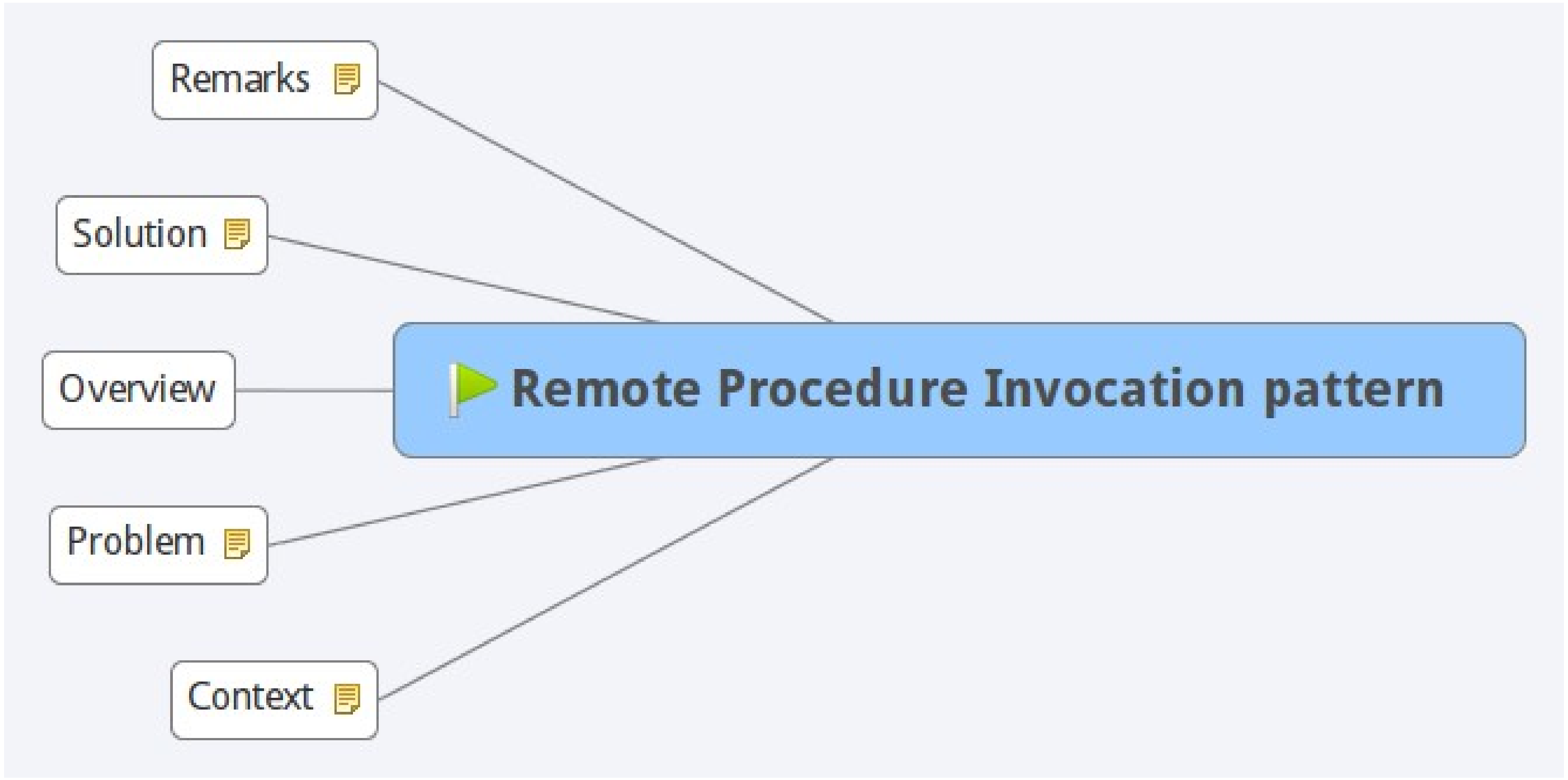
cc BY-NC-SA

Service Patterns



(cc) BY-NC-SA

Remote Procedure Invocation pattern



(cc) BY-NC-SA

Context

An enterprise has multiple applications that are being built independently, with different languages and platforms.

Need to share data and processes in a synchronous, timely way.

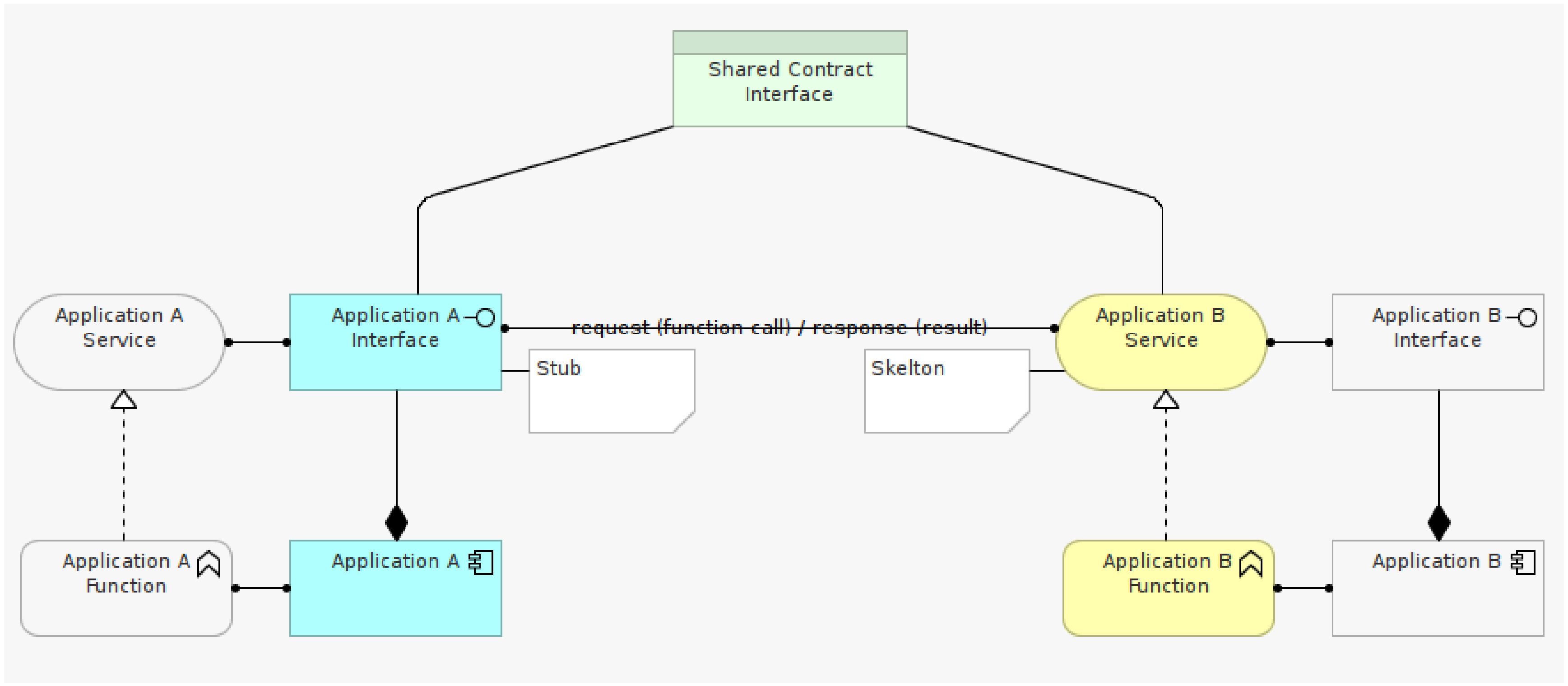
(cc) BY-NC-SA

Problem

How can I integrate multiple applications so that they work together and can exchange information?

cc BY-NC-SA

Overview



cc BY-NC-SA

Solution

Develop each application as a large-scale component with encapsulated behavior + data.

Provide a service and contract interface to allow other applications to interact with the running application.

Typically functions are invoked via a Web Service using Service-oriented model (SOAP/RPC) or a Resource-oriented model (WS-Literal).

(cc) BY-NC-SA

Remarks

Remote Procedure Invocation applies the principle of encapsulation to integrating applications.

If one application needs to acquire the data, or modify the data of another application, then it does so by making a direct call to the other application.

Each application maintains the integrity of the data it owns. Furthermore, each application can alter its internal data without having every other application be affected.

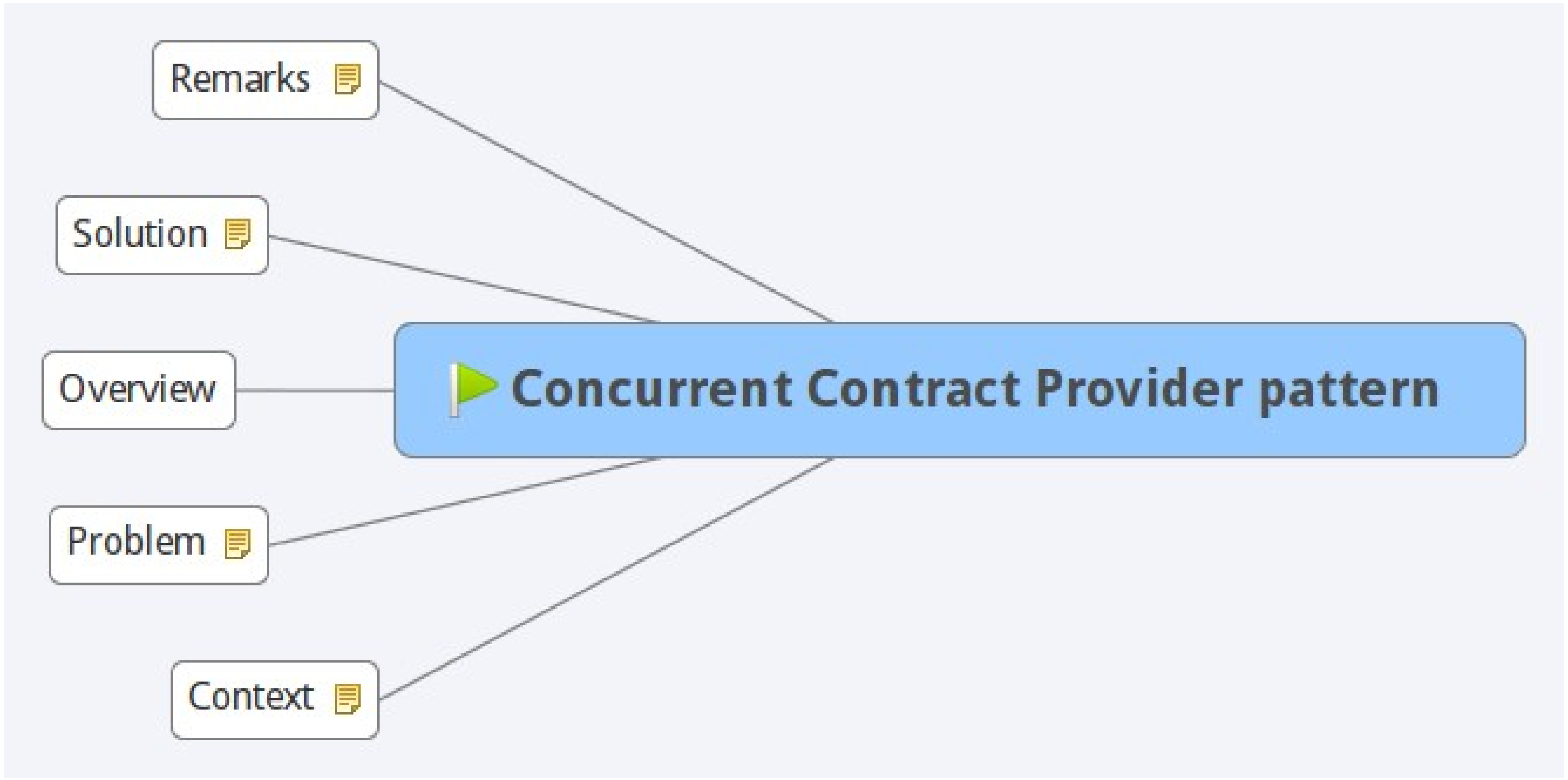
Distributed Object Integration is also known as instance-based collaboration because it extends the model of object-oriented computing to distributed solutions.

This type of synchronous interaction usually seems natural to architects, but it can result in a complex and tightly-coupled interaction model between the components.

Too often it is assumed that distributed component are up-and-running 24/7, instead of for failure by design.

(cc) BY-NC-SA

Concurrent Contract Provider pattern



(cc) BY-NC-SA

Context

Need for defining a method of centralizing sharing schemas across application boundaries to avoid having to manage redundant service definitions at risk of become out of date.

Need for a method allowing multiple applications with different versions of a same contract type to simultaneously consume the same service.

(cc) BY-NC-SA

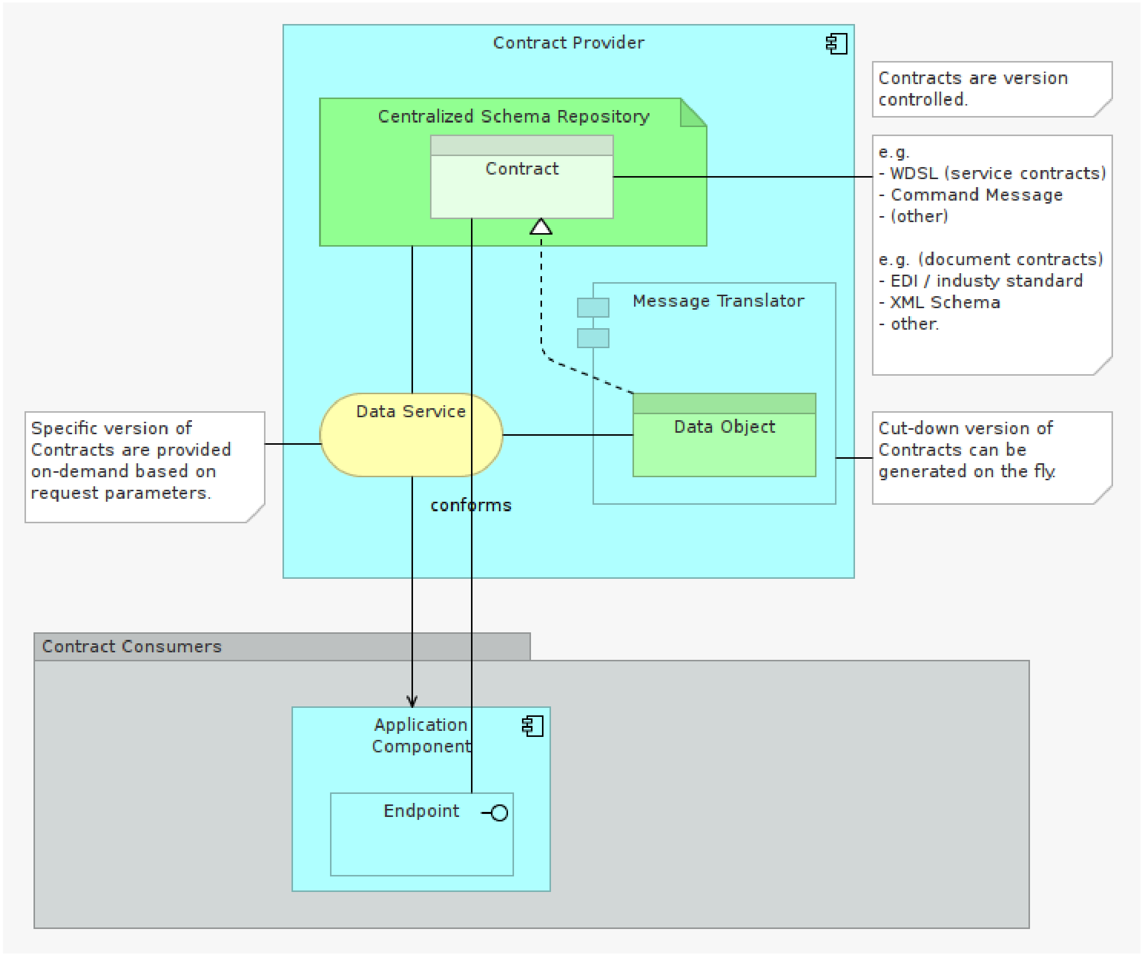
Problem

Similar data sets must be processed by services or applications with different capabilities, resulting in unwieldy service contracts or data schemas.

One unique version of a same service contract endpoint may not be suitable for all potential consumers, extensibility of is a key requirement.

(cc) BY-NC-SA

Overview


Contract Provider
Centralized Schema Repository
Contract
Message Translator
Data Service
Data Object
conforms
Contract Consumers
Application Component
Endpoint
Contracts are version controlled.
e.g.
- WDSL (service contracts)
- Command Message
- (other)
e.g. (document contracts)
- EDI / industy standard
- XML Schema
- other.
Specific version of Contracts are provided on-demand based on request parameters.
Cut-down version of Contracts can be generated on the fly.

BY-NC-SA

Solution

Centralize Define data schemas as entities in a persistent Contract Repository which is separated from the distributed components.

Any contract-first architecture, regardless of implementation technology (JMS, SOAP, other) in which more than one system will transmit, transform,

process, or store data can use the Contract Provider component to obtain the appropriate version of a contract, function of the target destination.

(cc) BY-NC-SA

Remarks

Multiple contracts may exist for the same service, each with a different level of abstraction than the others in the same group, to fit corresponding

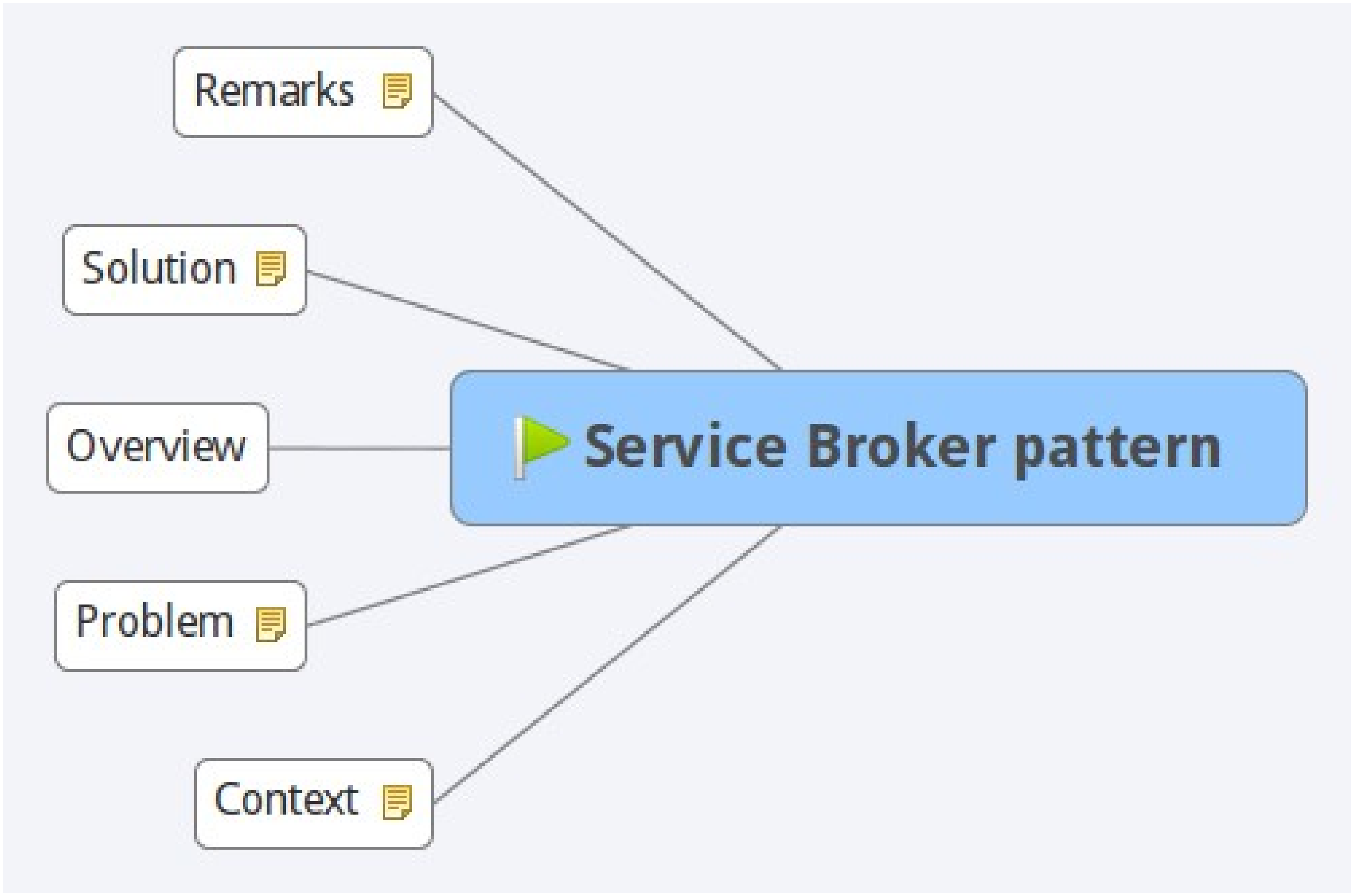
service level agreements or to accommodate legacy systems.

Contract definitions can be persisted on a file system or in a document-oriented data store. Caching is an option to make requests to data contracts as efficient as possible.

In complex implementations of this pattern, contracts can be generated on-demand based on an underpinning rules database.

(cc) BY-NC-SA

Service Broker pattern



(cc) BY-NC-SA

Context

Instead of being built from scratch, Application A needs to be build from a collection of existing services distributed across multiple servers.

Implementing Application A is complex because of the difficulties in getting existing systems to interoperate - how they will connect to each and how they will exchange information - as well as the availability of their component services.

cc BY-NC-SA

Problem

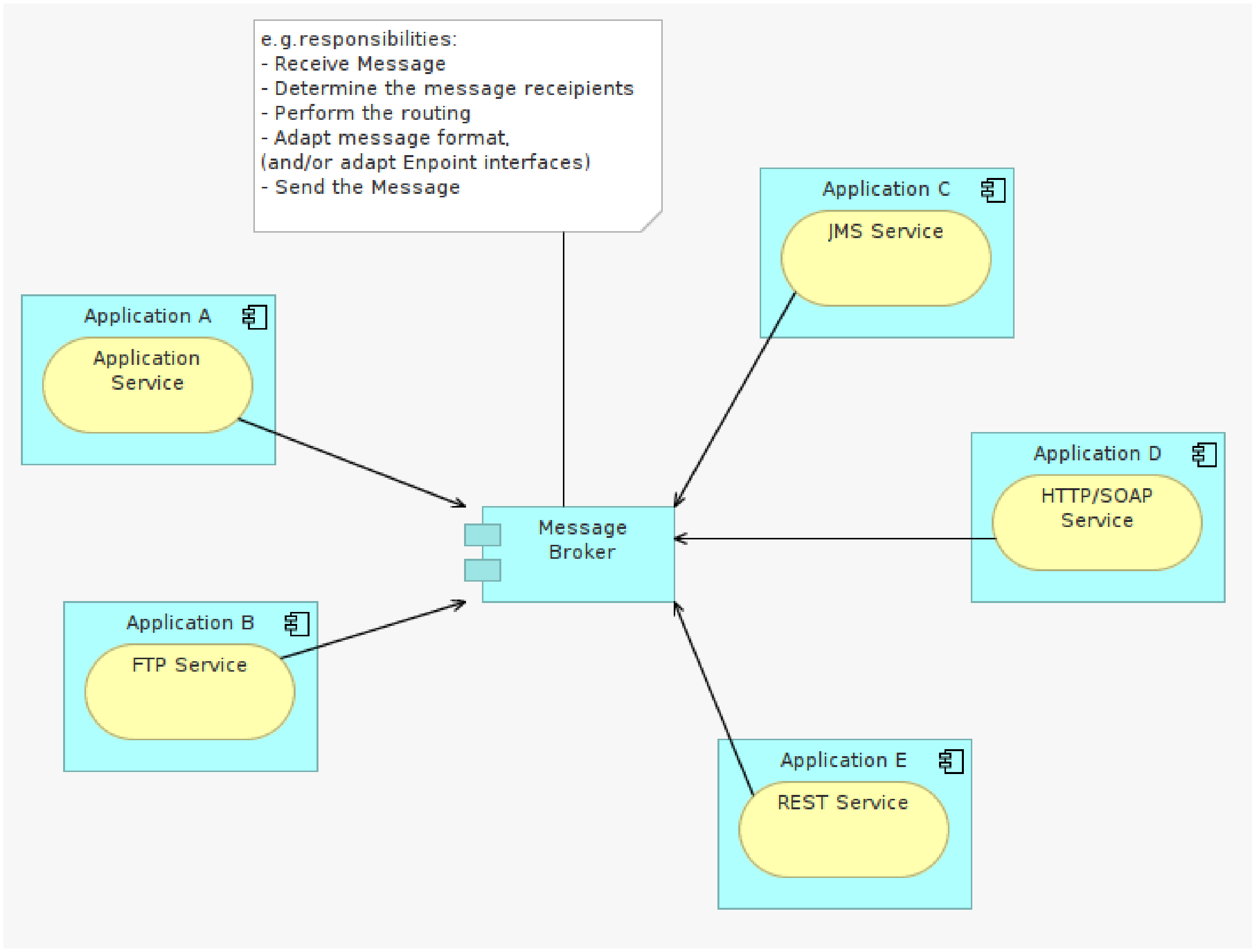
How can you decouple the destination of a message from the sender and maintain central control over the flow of messages?

How do you integrate applications without enforcing a common interface and also allow each application to initiate interactions with several other applications?

How do you structure& chains distributed components to form a cohesive application logic, without having to worry about the nature and location of service providers, making it easy to dynamically change the bindings/dependencies over time?

(cc) BY-NC-SA

Overview



(cc) BY-NC-SA

Solution

Integrate the composite services provided by distributed application using Message Broker pattern.

A message broker is a physical component that handles the communication between applications.

The Message Broker pattern separates the consumer of services (clients) from the provider of services (servers) by inserting an intermediary, commonly referred as a Broker.

Instead of communicating with each other, applications communicate only with the message broker.

BY-NC-SA

Remarks

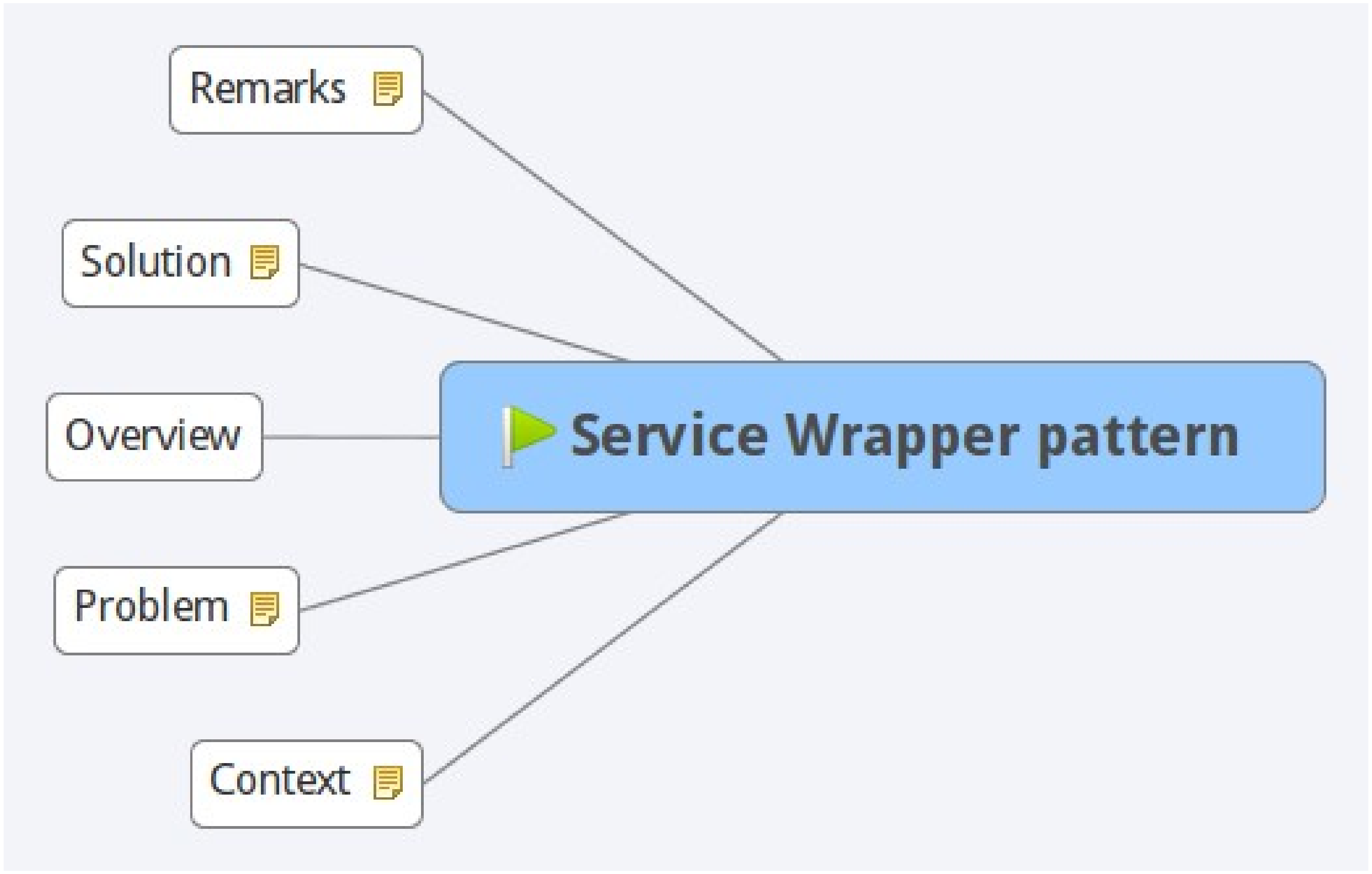
When a client needs a service, it queries a Broker via a service interface. The Broker then forwards the client service request to the eligible target server that must process the request.

An application sends a message to a Message Broker and specifies the the logical name of the receivers.

The message broker looks up applications that are registered under the logical name and then passes (i.e. re-route) the Message to them.

BY-NC-SA

Service Wrapper pattern



(cc) BY-NC-SA

Context

Need for Encapsulate a legacy service API inside a generic, modern stateless service exposed to distributed applications.

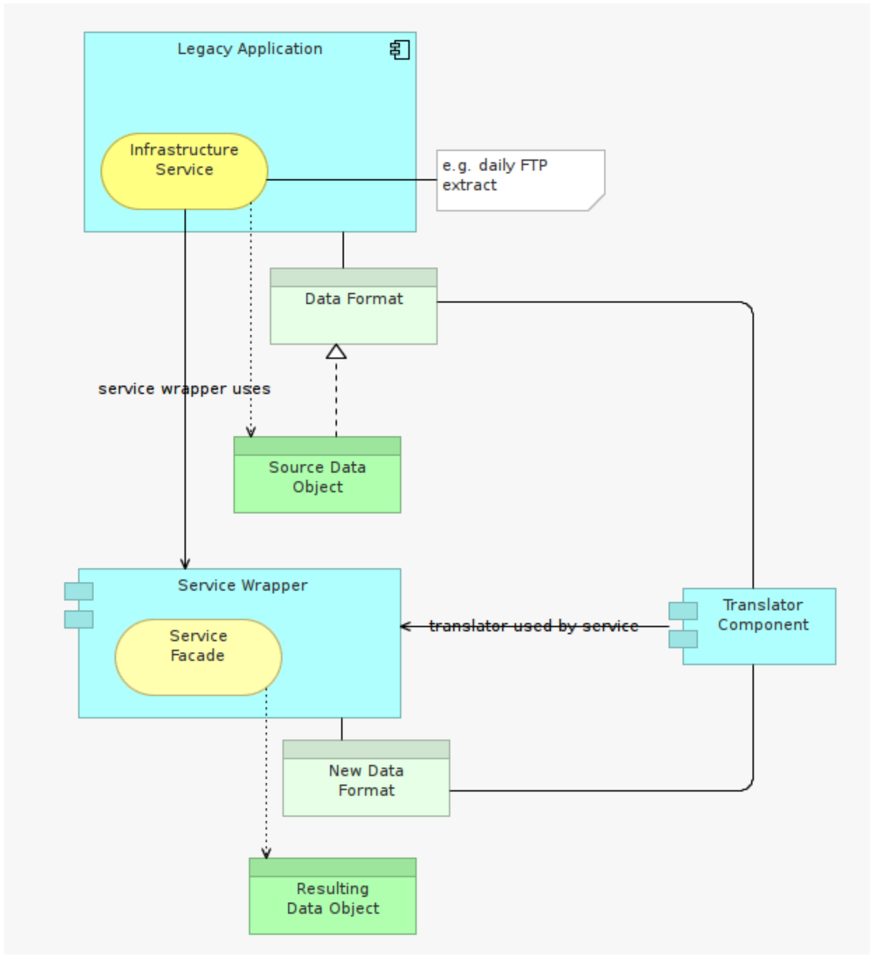
(cc) BY-NC-SA

Problem

Legacy systems may offer limited service capabilities, or their only interface with other applications may be through file data exchanges only.

BY-NC-SA

Overview



BY-NC-SA

Solution

Wrap the inter-operation mechanisms within a service facade that operates with the legacy system and exposes a normalized SOA interface to new consumers.

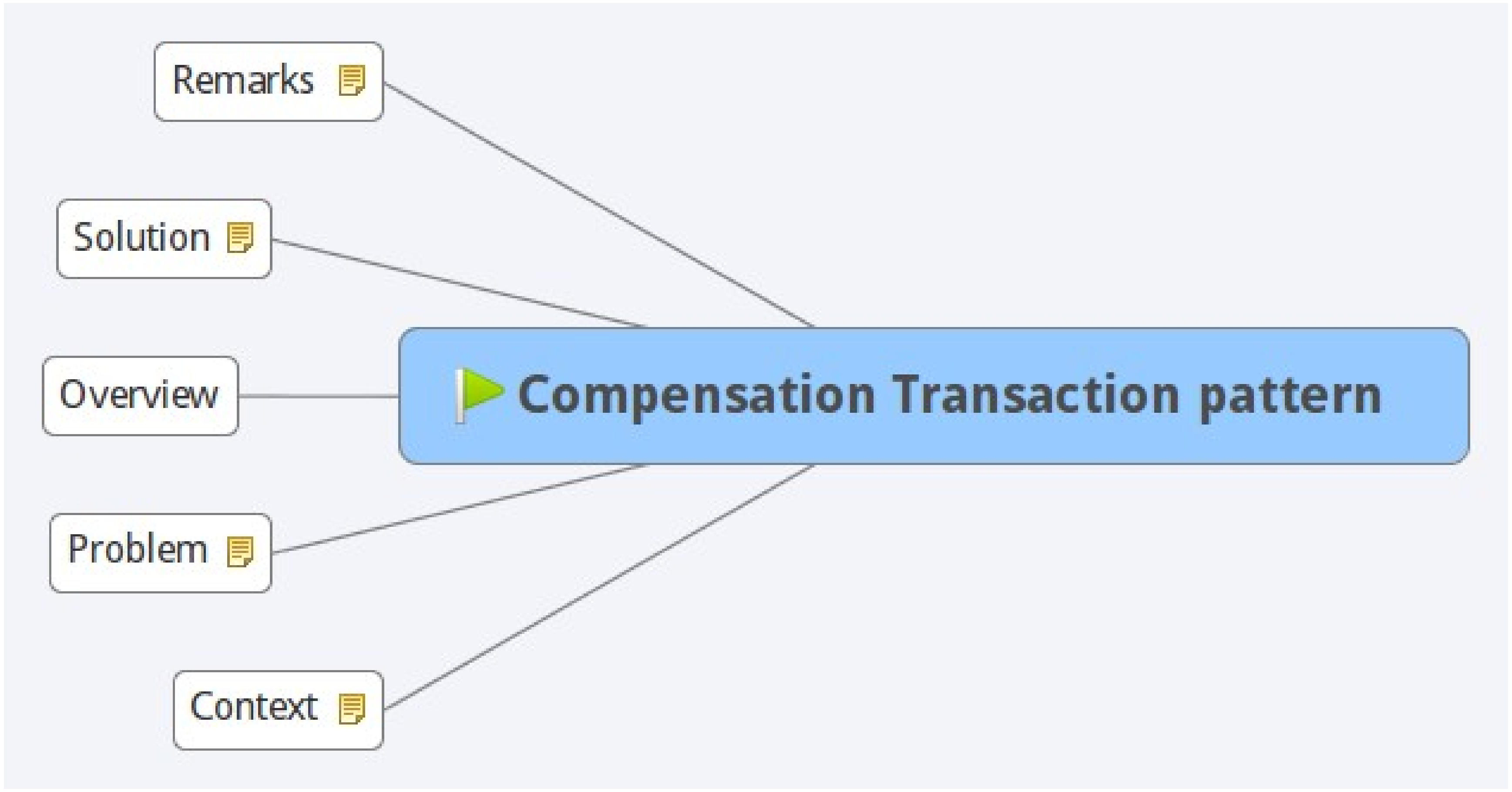
(cc) BY-NC-SA

Remarks

Also referred as Service Interface pattern. The Service Wrapper pattern is often implemented as a read-only capability.

(cc) BY-NC-SA

Compensation Transaction pattern



cc BY-NC-SA

Context

Distributed components inevitably increase the chances of failure in an architecture.

Two or more processes, each accessed via a service, possibly running concurrently across multiple systems, need to complete successfully for the overall transaction to succeed.

If one (or more) of the related services fail, all the services associated with it and the application response must roll-back to their previous state for maximum application integrity.

Need for a mechanism to coordinate multiple run-time activities which together comprise a single holistic service, with guaranteed completion or roll-back capability.

BY-NC-SA

Problem

Classic Atomic Transaction (ACID) models are impractical for long-running conversations between distributed components.

- Using ACID with long running conversations increase the time to commit a transaction, locking resources for prolonged periods of time (e.g. database table locks).

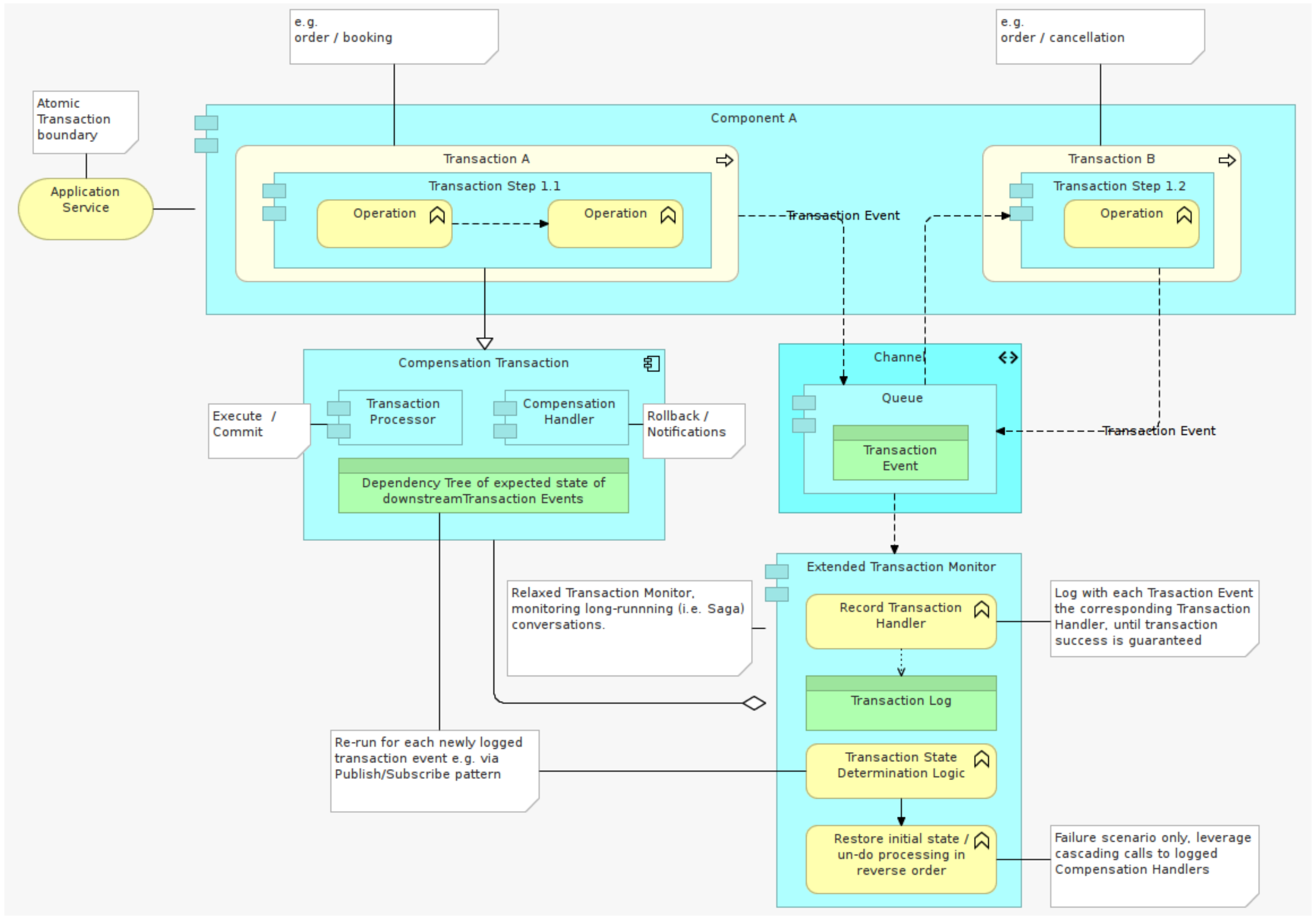
- Using ACID with long-running conversations created temporal coupling between components because the root coordinator of the transaction ultimately drives all the transactional resources through the 2 phase-commit protocol.

In a failure scenario, how to avoid compromising the integrity of the data underlying an application?

How can a rollback capability be propagated across services?

(cc) BY-NC-SA

Overview


e.g.
order / booking
e.g.
order / cancellation
Atomic Transaction boundary
Application Service
Component A
Transaction A
Transaction Step 1.1
Operation
Operation
Transaction Event
Transaction B
Transaction Step 1.2
Operation
Compensation Transaction
Transaction Processor
Compensation Handler
Execute / Commit
Rollback / Notifications
Dependency Tree of expected state of downstreamTransaction Events
Channel
Queue
Transaction Event
Transaction Event
Extended Transaction Monitor
Relaxed Transaction Monitor, monitoring long-runnning (i.e. Saga) conversations.
Record Transaction Handler
Log with each Trasaction Event the corresponding Transaction Handler, until transaction success is guaranteed
Transaction Log
Transaction State Determination Logic
Re-run for each newly logged transaction event e.g. via Publish/Subscribe pattern
Restore initial state / un-do processing in reverse order
Failure scenario only, leverage cascading calls to logged Compensation Handlers

(cc) BY-NC-SA

Solution

Wrap all service calls in a compensation-based service transaction (also referred as extended transaction).

Ensure all services provide a rollback feature (referred as a compensation handler) that resets all actions if the parent business task cannot be successfully completed. A compensation undoes the effect of a transaction.

Log the successes/failures and related compensation handlers at each progression of the transaction logic, within each service, using an extended transaction monitor component relaxing the ACID (possibly a transaction manager).

Granular services may be wrapped in another service that provides integrity checks and ensures successful completion or graceful degradation, if any, if the granular services fails.

(cc) BY-NC-SA

Remarks

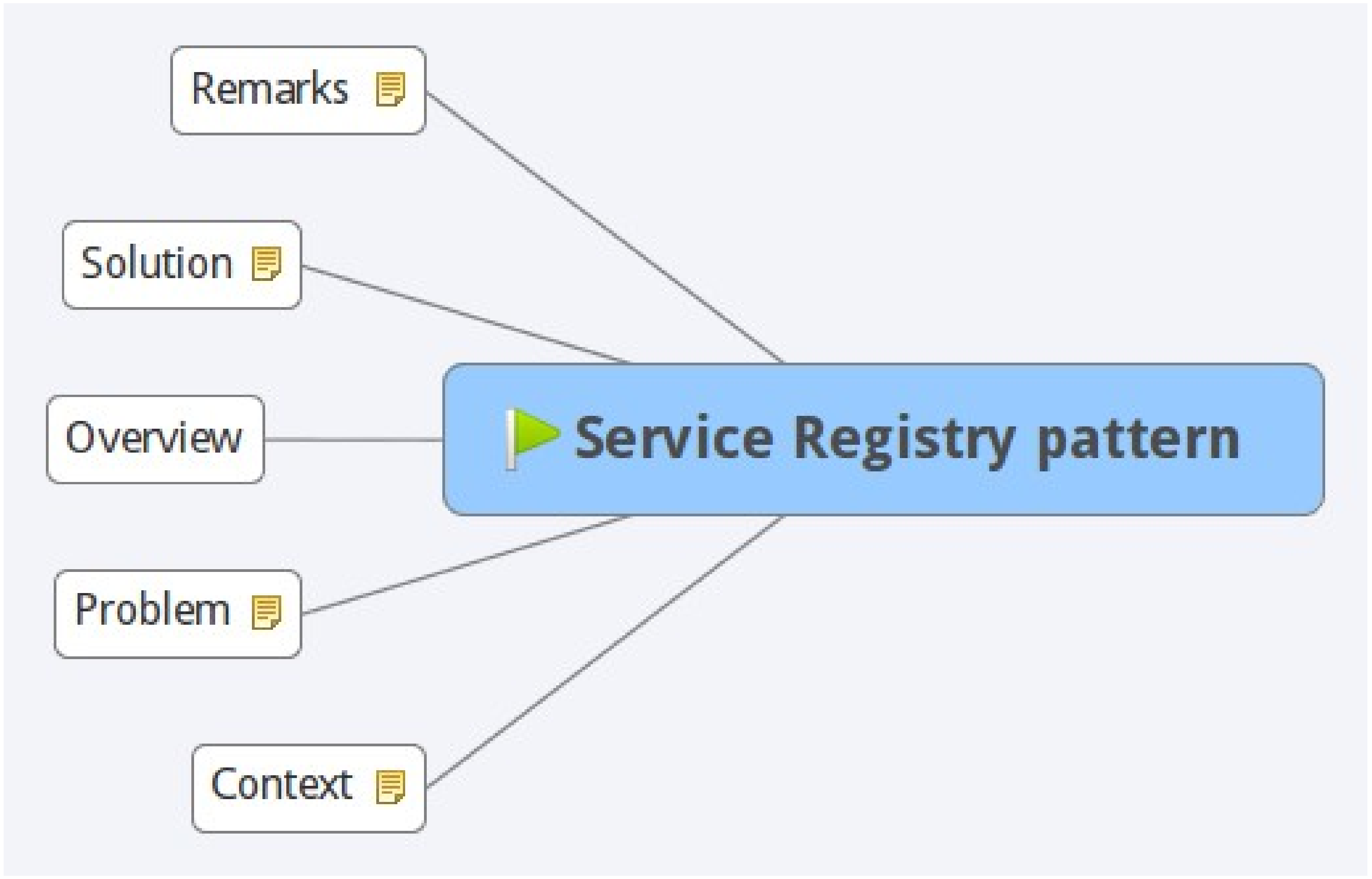
Extended Transacted activities can consumer resources to preserve compensation handlers for each granular service in case roll-back is necessary (but much cheaper resource-wise than ACID i.e. storing original state at each transaction step).

Note that in distributed computing architectures not all transactions can be rolled back (for example sending an Email).

This pattern can make use of a commercial Transaction Managers at risk of potential vendor lock-in.

(cc) BY-NC-SA

Service Registry pattern



(cc) BY-NC-SA

Context

Clients of a service need to determine the location of a service instance to which to send requests to.

Using an inversion of control (IOC) also referred as dependency injection is not an option because of the heterogeneity of the distributed components at play in the architecture.

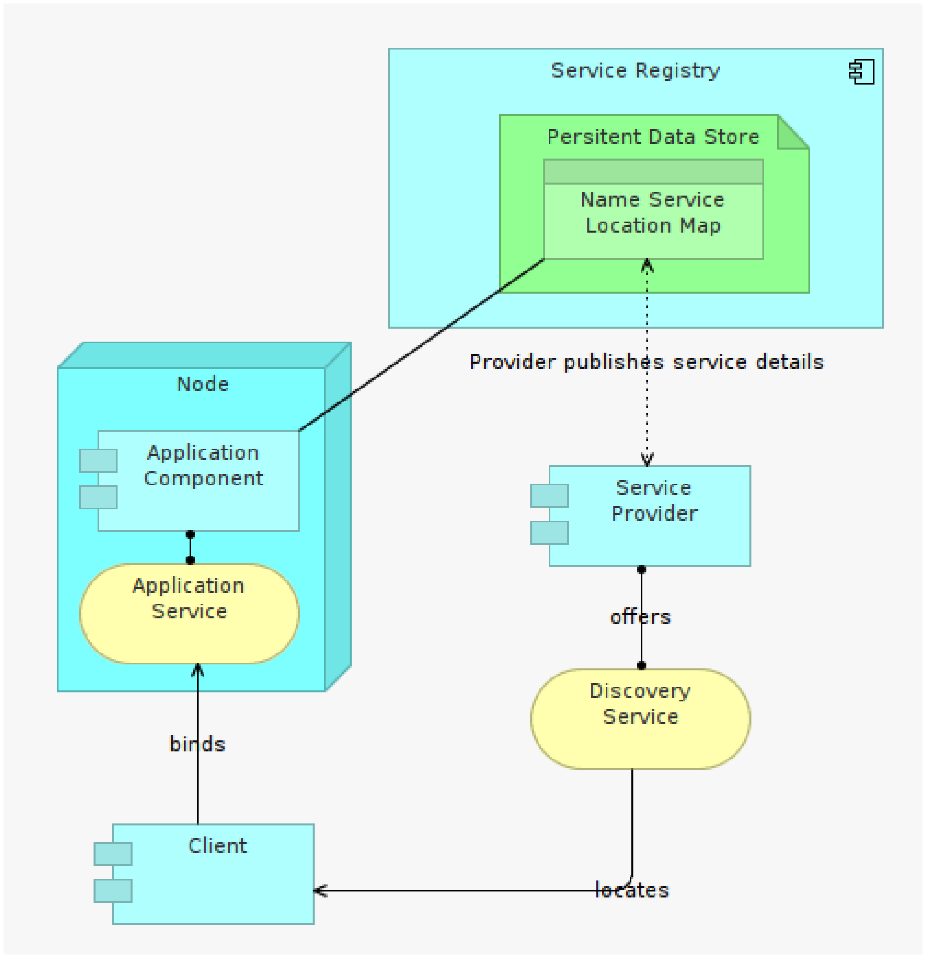
(cc) BY-NC-SA

Problem

How do clients of a service (in the case of Client-side discovery) and/or routers (in the case of Server-side discovery) know about the available instances of a service?

BY-NC-SA

Overview



BY-NC-SA

Solution

Implement a service registry, which is a centralized service locator supported by a persistent data store registering services, their instances and their locations.

Service instances are registered with the service registry on startup and deregistered on shutdown.

Client of the service and/or routers query the service registry to find the available instances of a service.

A Service Registry pattern provides a centralized point of control, and act as a cache that eliminates redundant lookups.

(cc) BY-NC-SA

Remarks

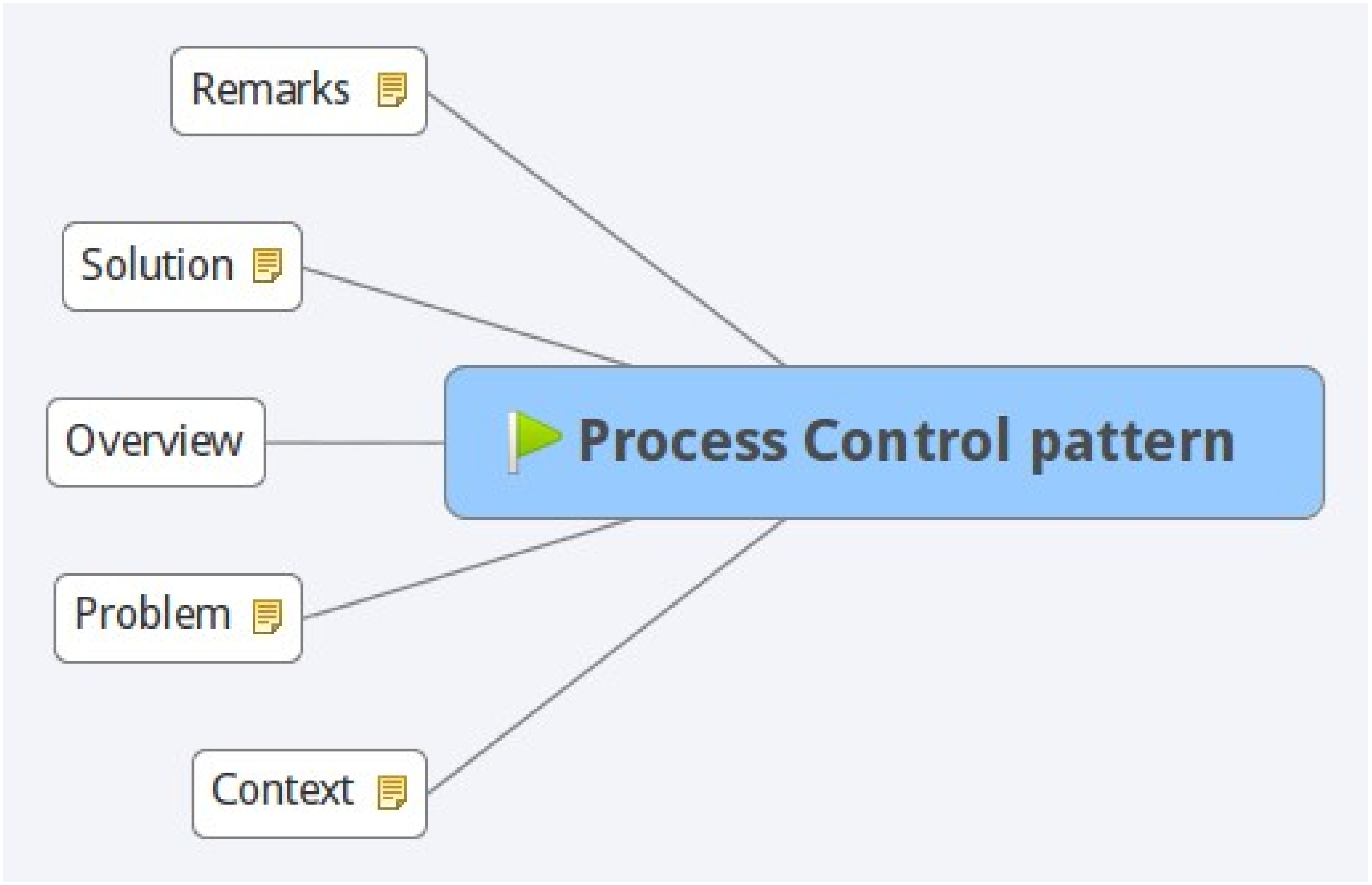
Every user of a service registry must have a dependency to the service locator. The locator can hide dependencies to other implementations, but you do need to see the locator.

Note: The service registry can become a single point of failure, but it can be scaled, cached, or its directory get eventually replicated across distributed components (not unlike DNS resolution/replication).

Example: Apache Zookeeper

cc BY-NC-SA

Process Control pattern



BY-NC-SA

Context

Need for an application of combining two or more non-sequential, inter-dependent processing steps.

Multiple services invocations are required to complete an operation and are known at design time, but the sequence of processing steps may vary depending on decision rules applied to the results received.

BY-NC-SA

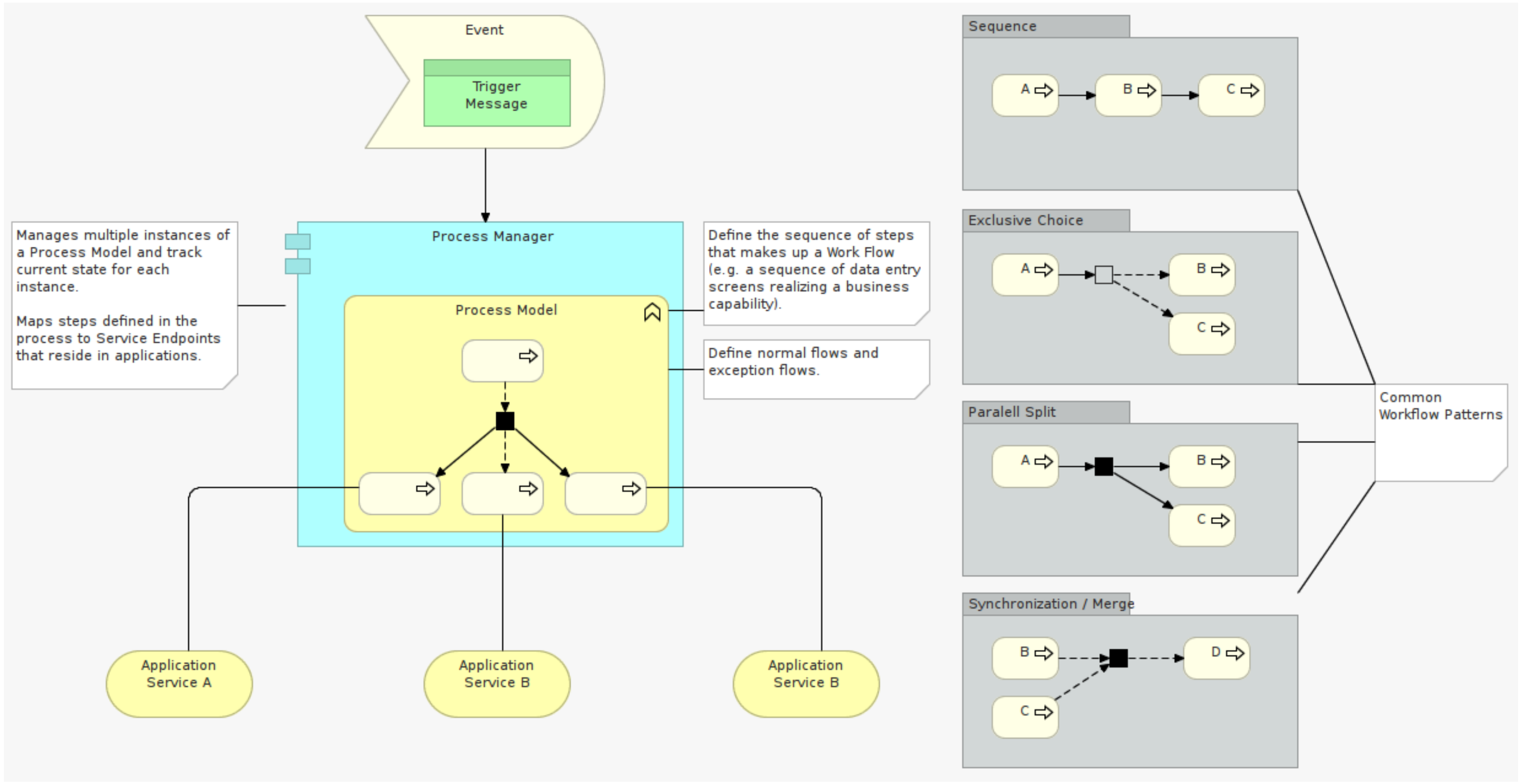
Problem

How do you coordinate the execution of a long-running business function that spans multiple disparate applications?

How do we route logic output through multiple business process steps?

(cc) BY-NC-SA

Overview


Event
Trigger Message
Process Manager
Process Model
Manages multiple instances of a Process Model and track current state for each instance.
Maps steps defined in the process to Service Endpoints that reside in applications.
Define the sequence of steps that makes up a Work Flow (e.g. a sequence of data entry screens realizing a business capability).
Define normal flows and exception flows.
Application Service A
Application Service B
Application Service B
Sequence
A
B
C
Exclusive Choice
A
B
C
Paralell Split
A
B
C
Synchronization / Merge
B
C
D
Common Workflow Patterns

(cc) BY-NC-SA

Solution

Define a business process model that describes the individual steps that make up the complex business function.

Use a central processing unit, a Process Manager (or Process Engine), to maintain the state of the sequence and determine the next processing step based on intermediate results.

Create a separate process manager component that can interpret multiple concurrent instances of this model and that can interact with the existing applications to perform the individual steps of the process.

After one application completes its business function, the process manager determines which function to execute next based on the state of the process instance.

Therefore, each participating application can operate individually without any knowledge of the sequence of steps defined in the process model.

(cc) BY-NC-SA

Remarks

Orchestration of processes involves messages sent to external systems and received from external systems.

These external systems implement the actions that make up the business process. At any time, there are likely to be many instances of the business process running at once, and any message that is received must be correlated with the correct instance of the business process that it was intended for.

Process Integration is commonly used to streamline the execution of a sequence of tasks. One popular application in the financial industry is the notion of straight-through processing (STP).

Frameworks:

- JBPM []http://www.jbpm.org/] BPM Engine
- DROOLS Decision Rules Engine (BRE)

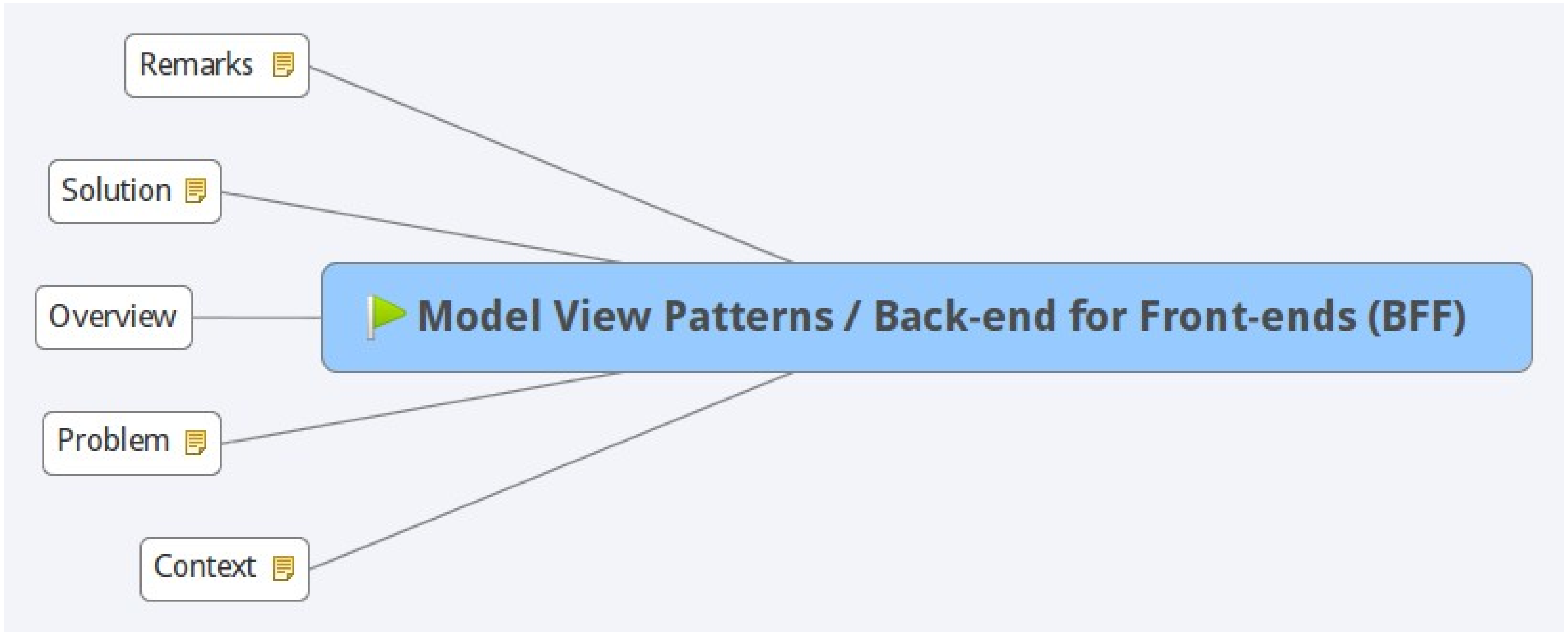
BY-NC-SA

Presentation Patterns

Presentation Patterns
Model View Patterns / Back-end for Front-ends (BFF)

(cc) BY-NC-SA

Model View Patterns / Back-end for Front-ends (BFF)



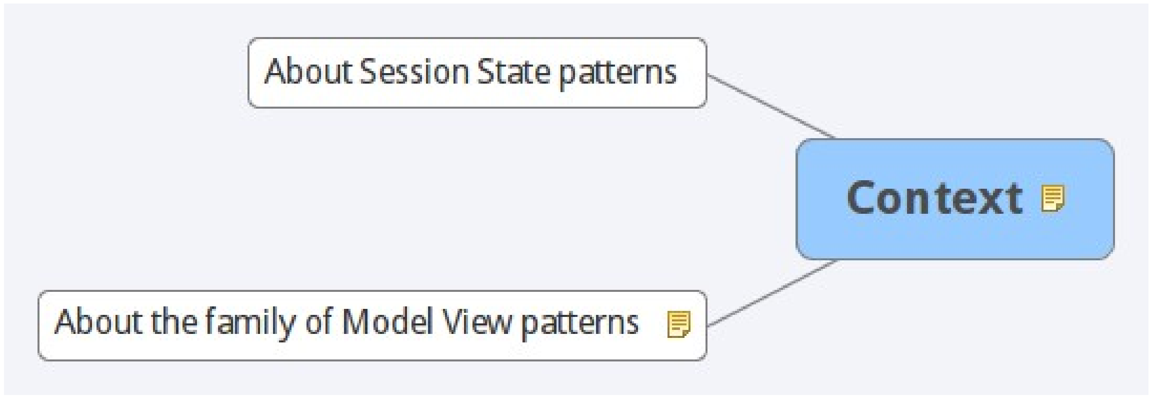
BY-NC-SA

Context

Based on our 1st lecture, we have reviewed typical statistics for Enterprise RIA implementing an insurance quoting system:

- Composed ~=300 visible Web UI screens
- Delivered to ~=1500 active users, up to ~=3000
- Supporting 15-20 business processes

The same EAA now needs to be adapted so to enable Quoting capabilities on Tablet devices and Smart phones.


About Session State patterns
Context
About the family of Model View patterns

(cc) BY-NC-SA

About the family of Model View patterns

Splits user interface interaction into three distinct roles: Views - Controllers/View-Models - Model.

Variations of server-side implementation:

- Controller/Model components using "Presentation Template" pattern on server side

- All Components on client-side with AJAX calls to Application API via services

Variations of mono-directional / bi-directional data binding on the client-side:

- Data binding between View and Model

- Data binding between View and View-Model

Both MVP and MVVM are derivatives of MVC. The key difference between both is the dependencies each layer has on the other layers, as well as how tightly bound they are with regard to each other.

Modern Web Development Frameworks Mobile or JS fully expose HTTP and provide excellent implementation of MVC, MVP and/or MVVP.

BY-NC-SA

Model

A Model represents domain-specific data or information a web application will be working with. Models hold information but don't handle behavior.

(cc) BY-NC-SA

Views

A View is the part of the application that users interact with. A view formats how data appears in the browser.

It contains the data bindings and events and behaviors, it is an interactive UI that represents the state of a Controller/View-Model.

Views are not responsible for handling state.

(cc) BY-NC-SA

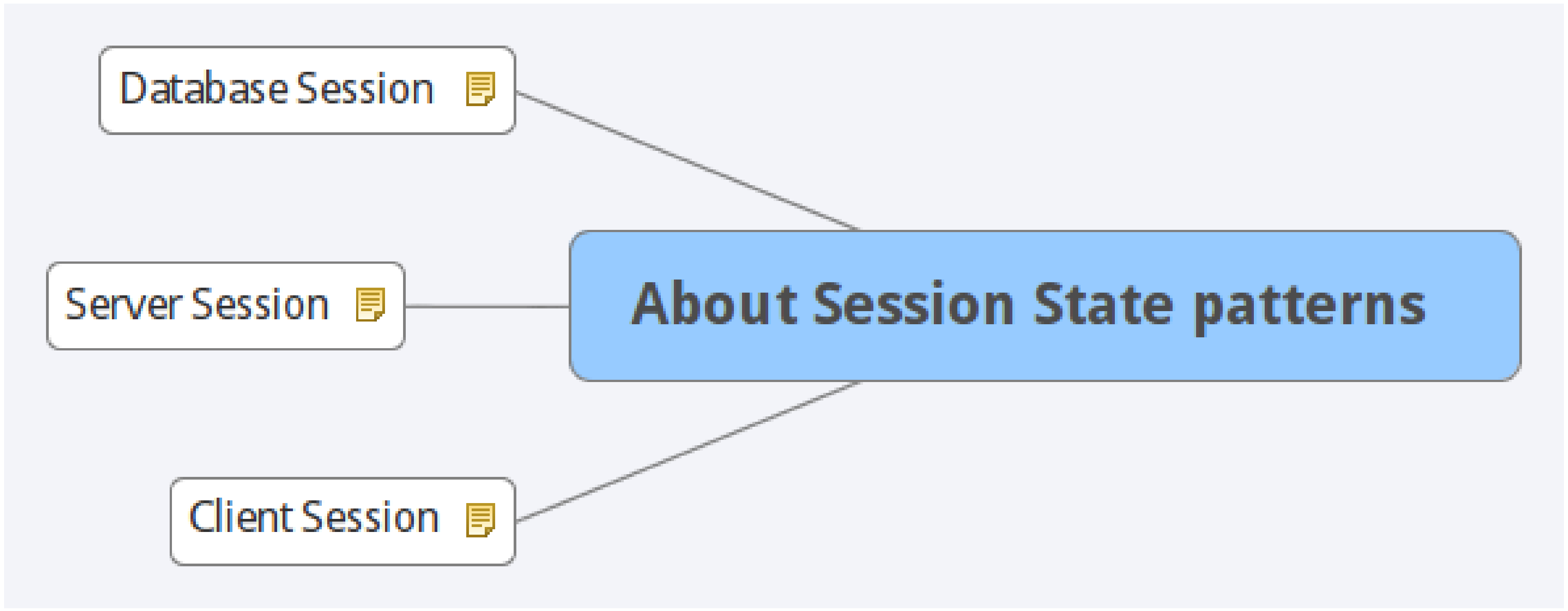
Controller / View-Model

A Controller / View-Model can be considered as a specialized Controller that acts as a data converter.

It changes Model information into View information, passing commands from View to Model, and manages state.

(cc) BY-NC-SA

About Session State patterns



(cc) BY-NC-SA

Session States

Client Session State: Stores session state on the client. Session data is not stored by the server but are added to each subsequent HTTP request, using the cookie mechanism.

Server Session State: Keeps the session state on a server system in a serialized form. This is the most common session pattern in use. Frameworks often provide this capability as a turn-key solution.

Database Session State: Need to stores session data as committed data in the database. A useful pattern to keep session persistent over long periods of time, and provide session capability Across heterogenous Devices (Mobile, Web, other).

BY-NC-SA

Problem

How to provide for mobile devices without having to rewrite application/domain logic?

How to maximize the re-use of application/domain logic regardless of the client device used by users?

How to persist state between multiple heterogenous devices?

How to separate concerns between components in such a way that the an enterprise RIA composed of hundreds of screens and work flows remains maintainable?

(cc) BY-NC-SA

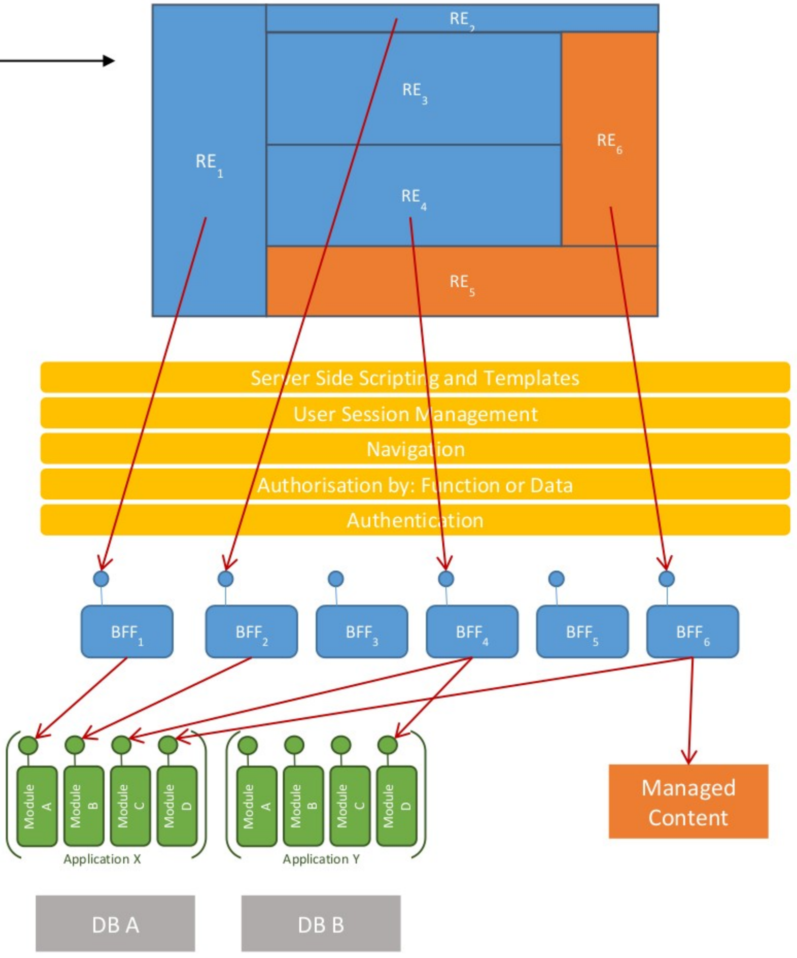
Overview

The application web pages comprise "real-estates" that align to respective BFFs.

Web applications have cross cutting concerns.

Applications are collections of services.

BFF, Backend for frontend, is a dedicated server side component whose code addresses the concerns for it client real estates.



A "micro release" comprises the changes in:
- a given real estate +
- BFF
- Modules on which the BFF depends.

(cc) BY-NC-SA

Solution

Consider architecting the application/domain logic of your enterprise portal using a micro-services architectural style.

Consider architecting the RIA of an enterprise portal around capabilities/features of the enterprise application using a Back-end for Front-end pattern.

Use a Back-end for Front-ends (BFF) pattern, in which a dedicated Component on the Server addresses the concerns for the Web client real-estate it corresponds to.

Use a Back-end for Front-ends (BFF) pattern, in which a dedicated Component on the Server addresses the concerns for the Client Device it corresponds to.

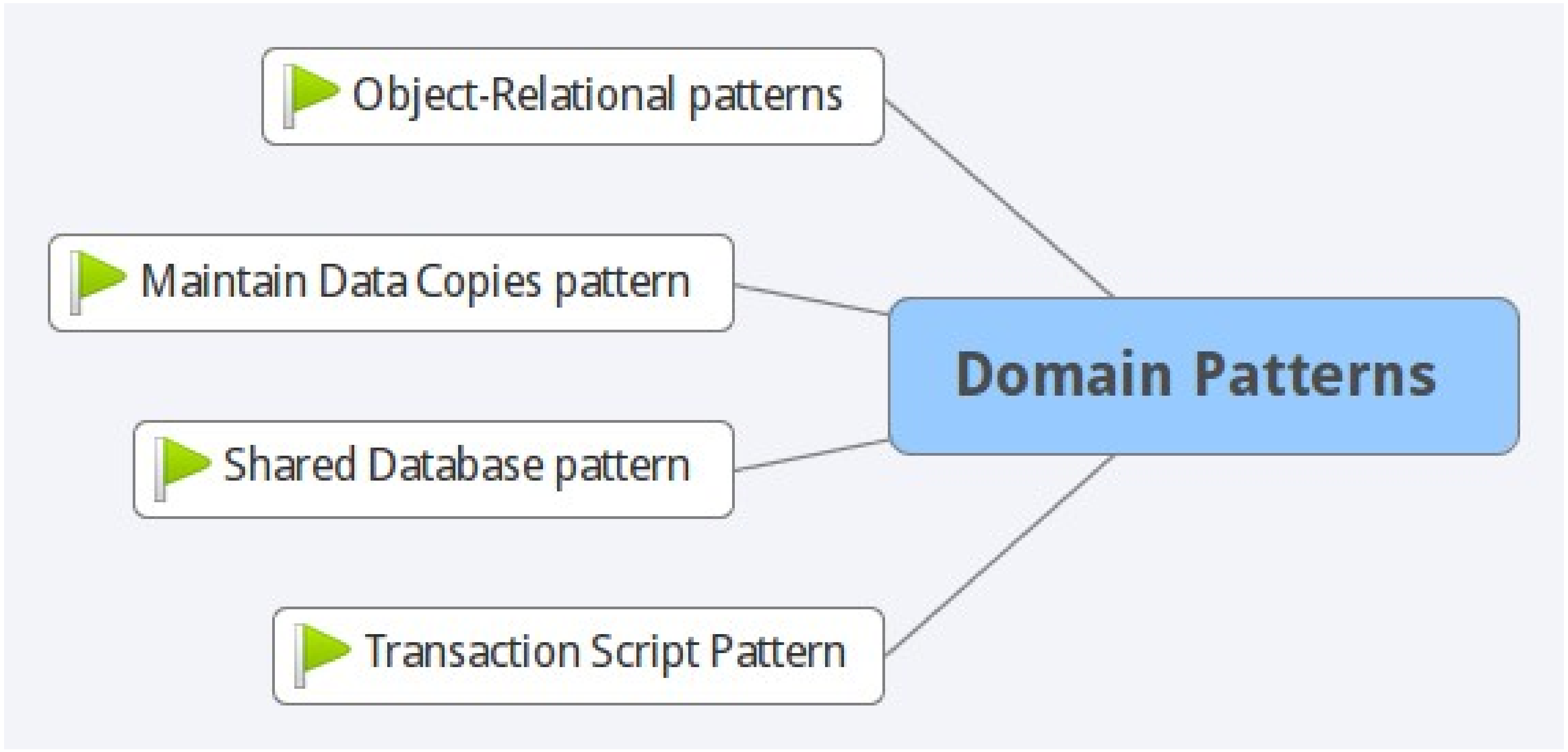
(cc) BY-NC-SA

Remarks

Visit the following link for more detail about this emerging pattern: [http://samnewman.io/patterns/architectural/bff/]

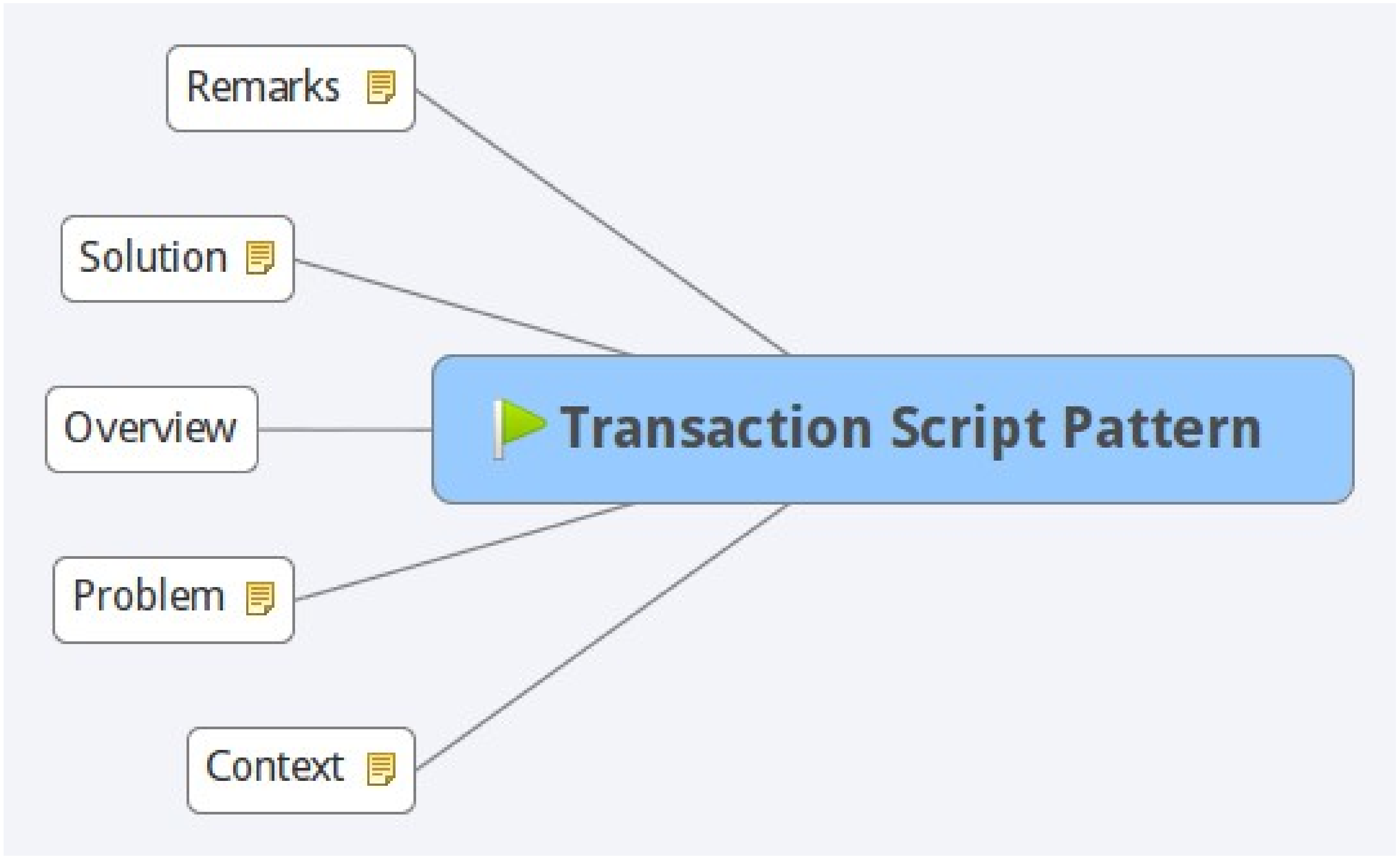
(cc) BY-NC-SA

Domain Patterns


Object-Relational patterns
Maintain Data Copies pattern
Shared Database pattern
Transaction Script Pattern
Domain Patterns

(cc) BY-NC-SA

Transaction Script Pattern



(cc) BY-NC-SA

Context

Each interaction between a client system and a server system contains a certain amount of logic.

Rules and logic describe many different cases and slants of behavior generating complexity.

cc BY-NC-SA

Problem

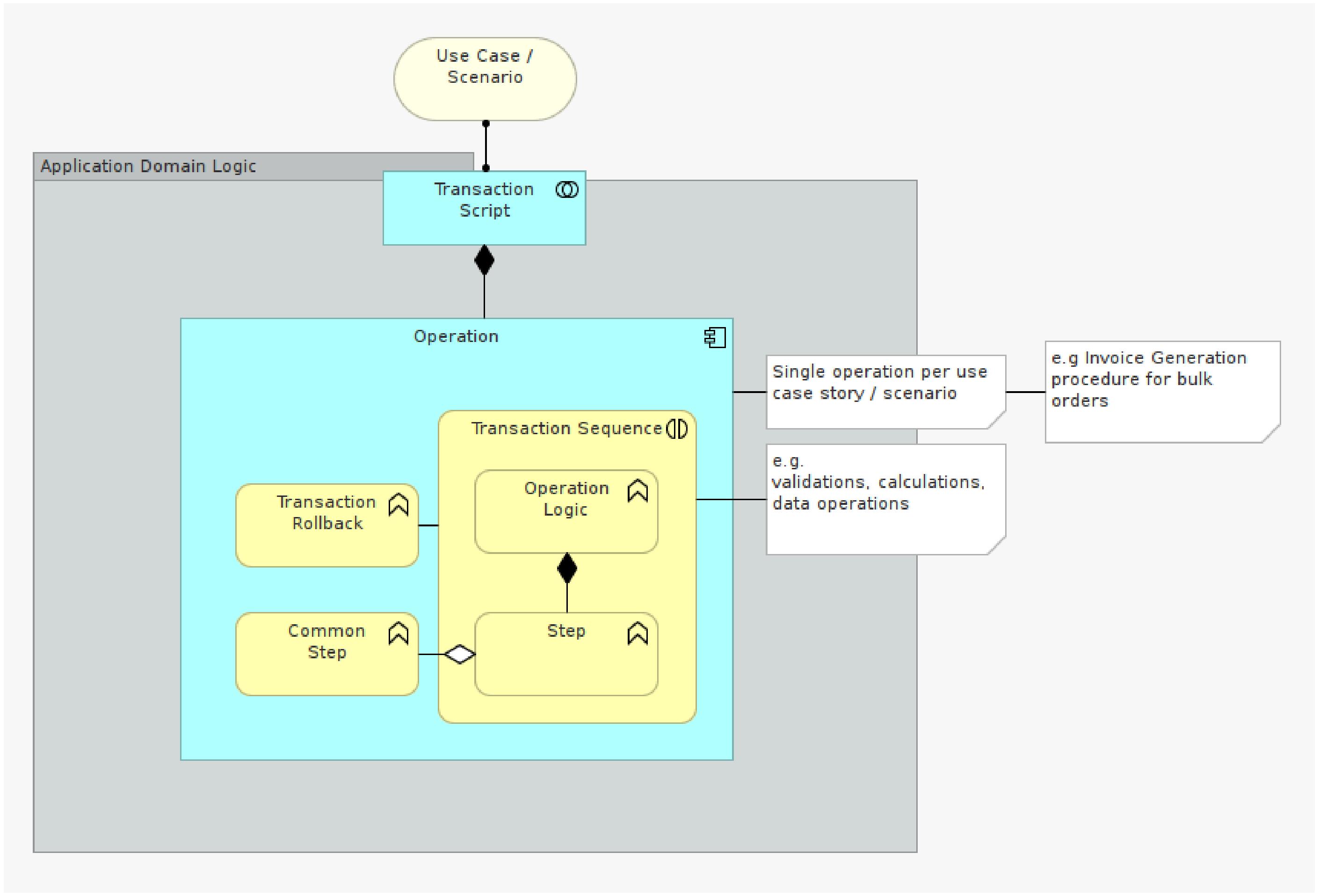
How to bring coherence, cohesion and consistency to application domain logic of a component/system?

How to breakdown and keep complexity under control as application grows?

How to revert back a transaction in the event that one of its constituting part isn't successful?

BY-NC-SA

Overview


Use Case / Scenario
Application Domain Logic
Transaction Script
Operation
Transaction Sequence
Transaction Rollback
Operation Logic
Common Step
Step
Single operation per use case story / scenario
e.g Invoice Generation procedure for bulk orders
e.g. validations, calculations, data operations

BY-NC-SA

Solution

A Transaction Script component organizes the corresponding logic primarily as a single operation, making downstream calls to other components, or directly to a data source.

Each scenario has its own corresponding atomic Transaction Script.

Each Transaction Script sequences operation logic step-by-step, method-by-method; method that can be roll-backed in case of failure.

Common steps may be broken into operations for purposes of re-use.

(cc) BY-NC-SA

Remarks

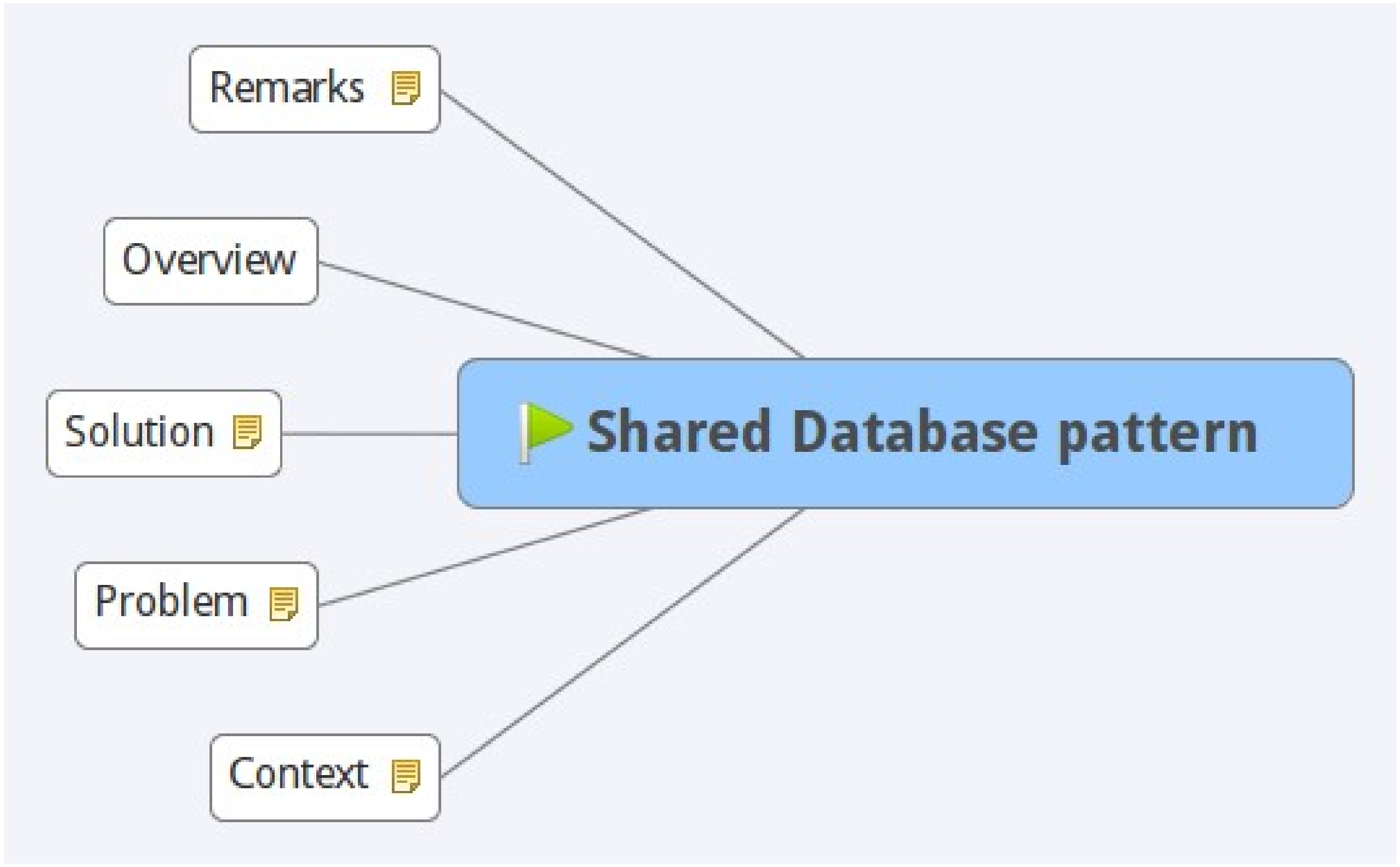
This pattern is a step toward the Domain Model pattern, simpler to implement.

Easy to implement, small/light, and can be refactoted into Domain Model pattern later.

Works well for small applications, clicks well with data source patterns, clicks well with Orchestration / Choregraphy patterns.

(cc) BY-NC-SA

Shared Database pattern



(cc) BY-NC-SA

Context

An enterprise has multiple applications that are being built independently, with different languages and platforms.

The enterprise needs information to be shared rapidly and consistently.

Data Assets are subject to strong regulatory requirements and must be managed in one place.

Decentralized data sources must reconcile (comply) with master data / authoritative sources.

(cc) BY-NC-SA

Problem

How do you integrate information systems that were not designed to work together?

How can I integrate multiple applications so that they work together and can exchange authoritative information?

(cc) BY-NC-SA

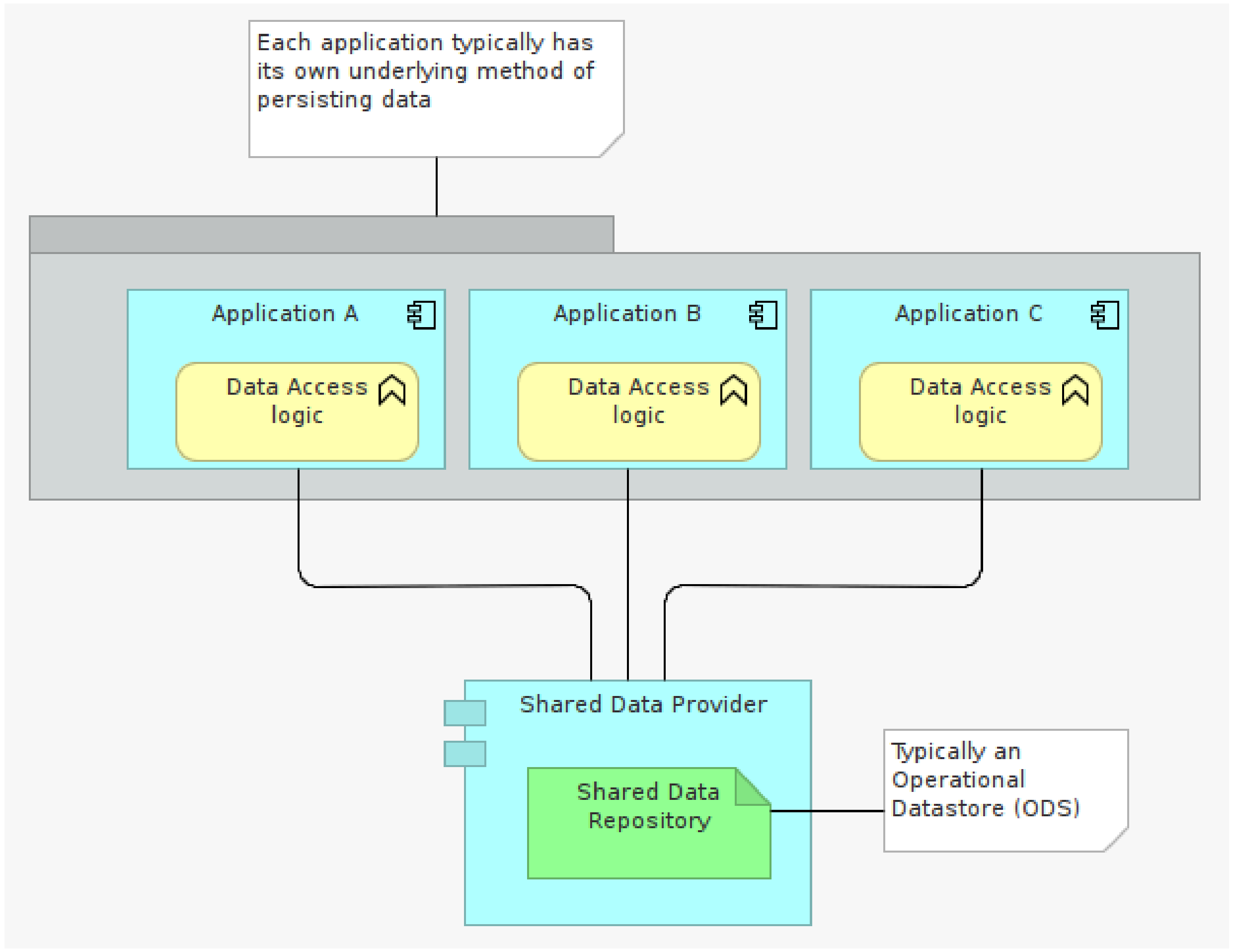
Solution

Integrate applications at the logical data layer.

Integrate applications by having them store their data in a single Shared Database.

CC BY-NC-SA

Overview



CC BY-NC-SA

Remarks

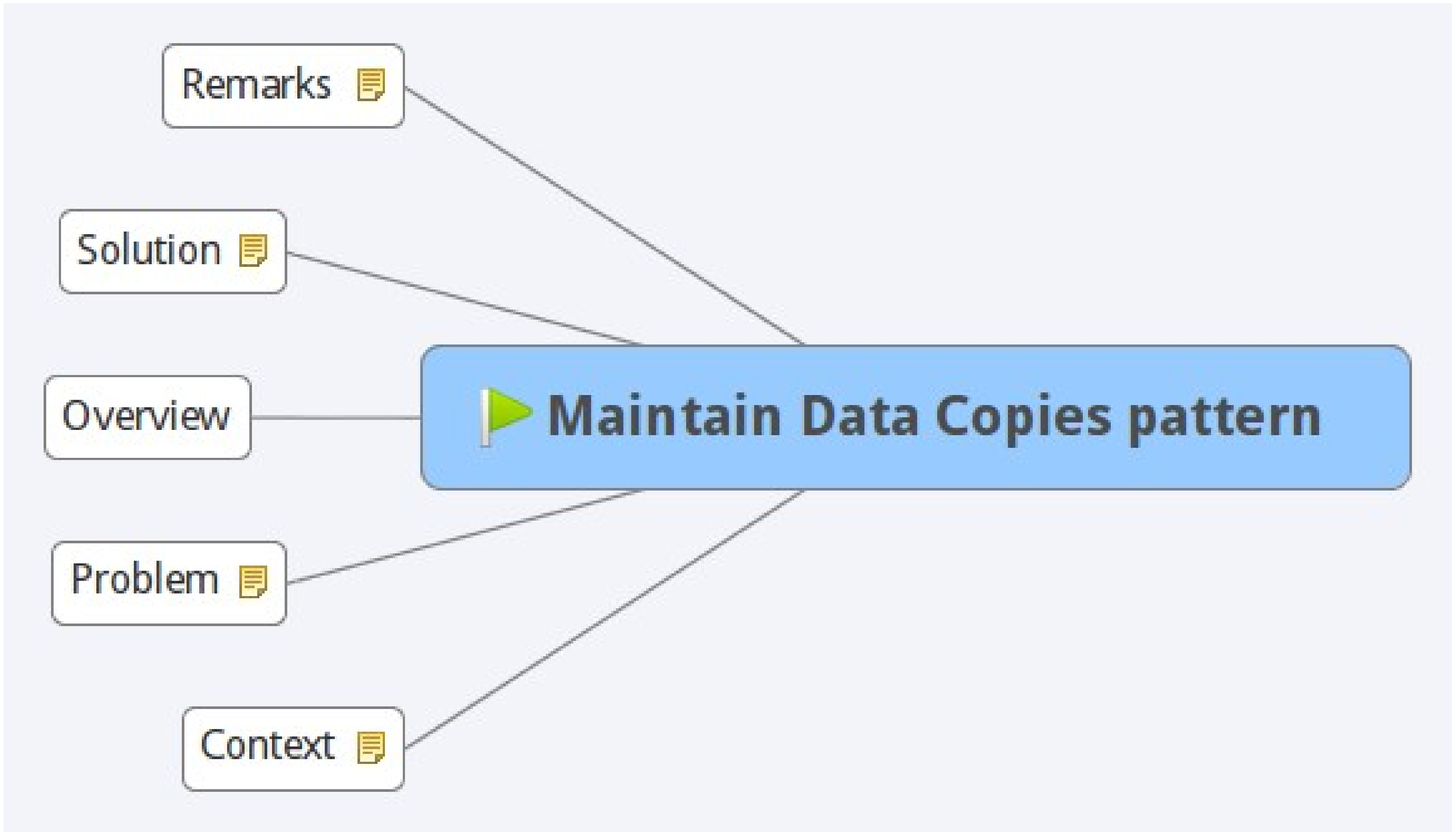
If a family of integrated applications all rely on the same database, then you can be pretty sure that they are always consistent all of the time.

If you do get simultaneous updates to a single piece of data from different sources, then you have transaction management systems that handle that about as gracefully as it ever can be managed.

Since the time between updates is so small, any errors are much easier to find and fix.

(cc) BY-NC-SA

Maintain Data Copies pattern



BY-NC-SA

Context

Need to have read-only copies of some OLTP data sets for heavy duty reporting purposes (i.e. complex query joins in a RDMBS for example).

Reporting activities must not impact the performance of source Transactional Systems.

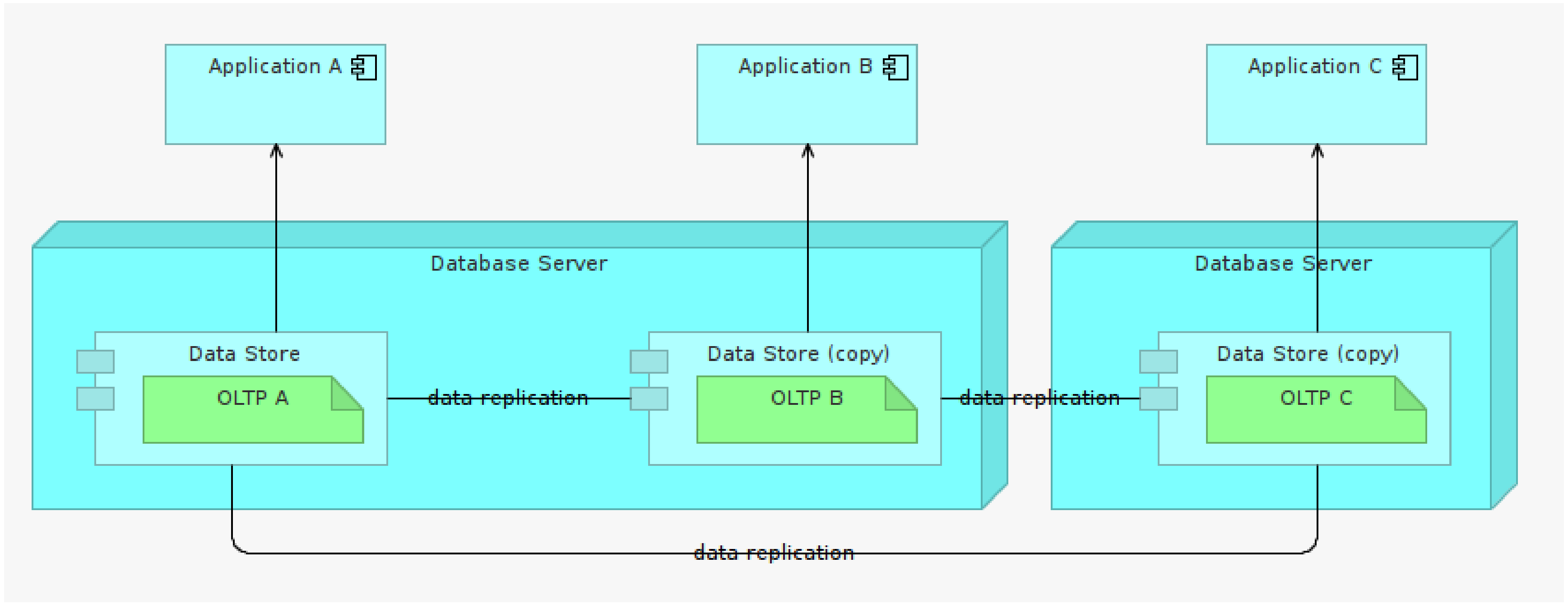
(cc) BY-NC-SA

Problem

How do you allow disparate application databases to share information without impacting performance of each other?

cc BY-NC-SA

Overview



(cc) BY-NC-SA

Solution

Have multiple applications access multiple copies of the same data. Maintain state integrity between copies.

Instead of sharing a single instance of a database between applications, make multiple copies of the database so that each application has its own dedicated store. Keep these copies synchronized, by copying data from one data store to the other.

This approach implies that each different data stores are slightly out of synchronization due to the latency that is inherent in propagating the changes from one data store to the next, referred as eventual consistency.

(cc) BY-NC-SA

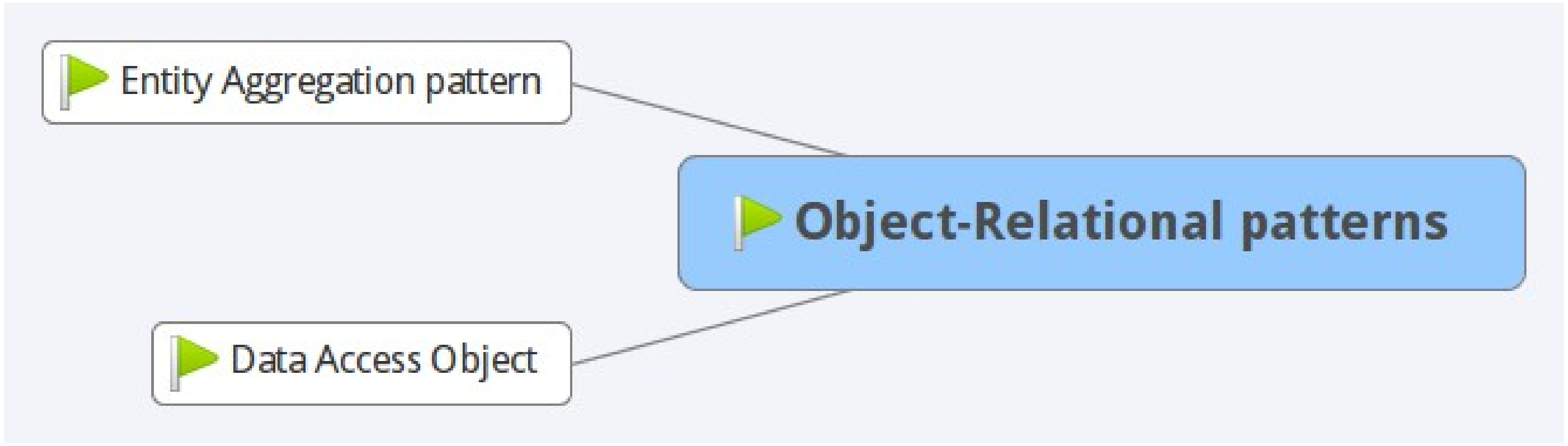
Remarks

The mechanisms involved in maintaining data copies are complex, e,g, Move Copy of Data

- Real-time Data Replication
- Master-Master Replication
- Master-Subordinate Replication
- Master-Master Row-Level Synchronization
- Master-Subordinate Snapshot Replication
- Capture Transaction Details
- Master-Subordinate Transactional Incremental

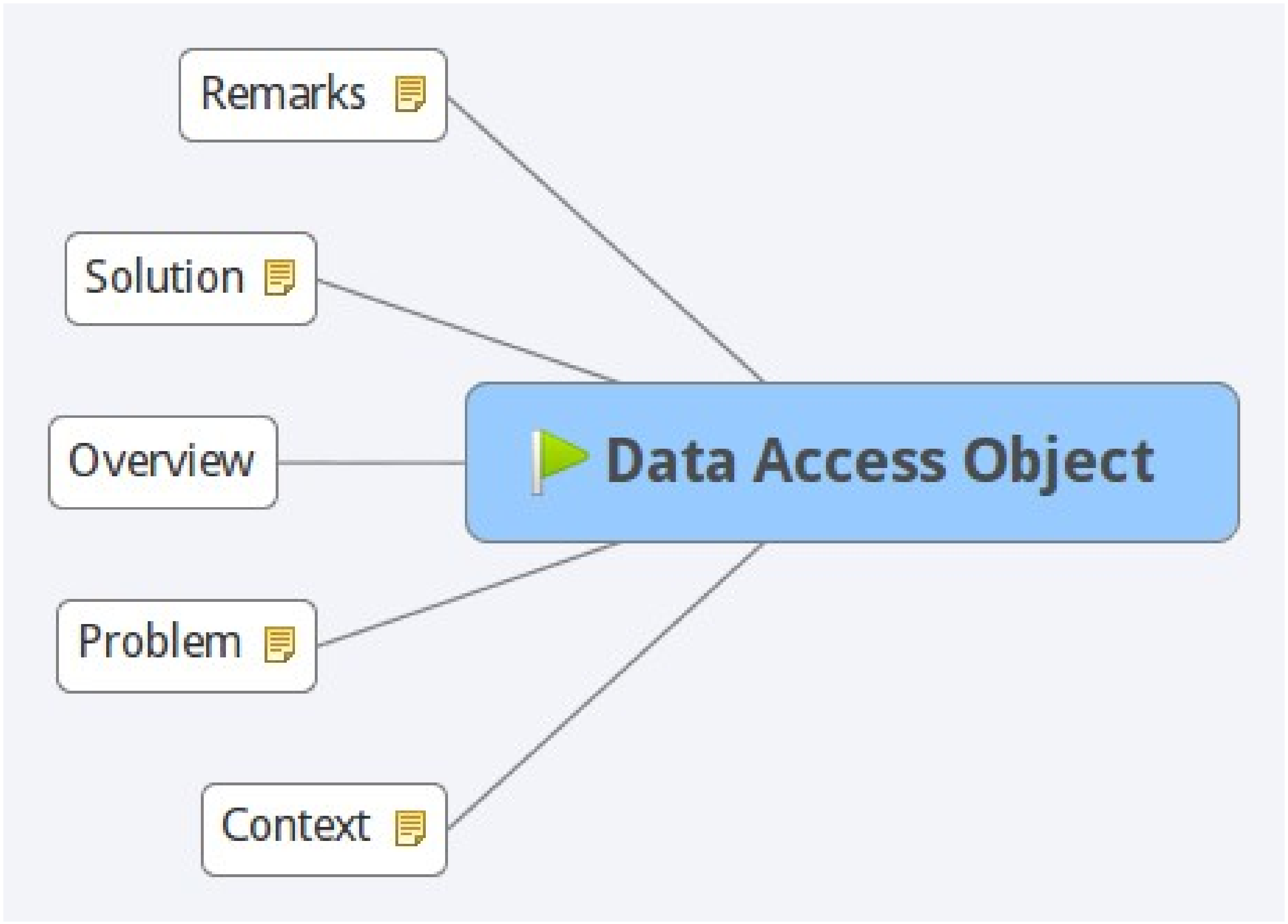
(cc) BY-NC-SA

Object-Relational patterns



BY-NC-SA

Data Access Object



(cc) BY-NC-SA

Context

Need to access data, where the source of the data can vary.

BY-NC-SA

Problem

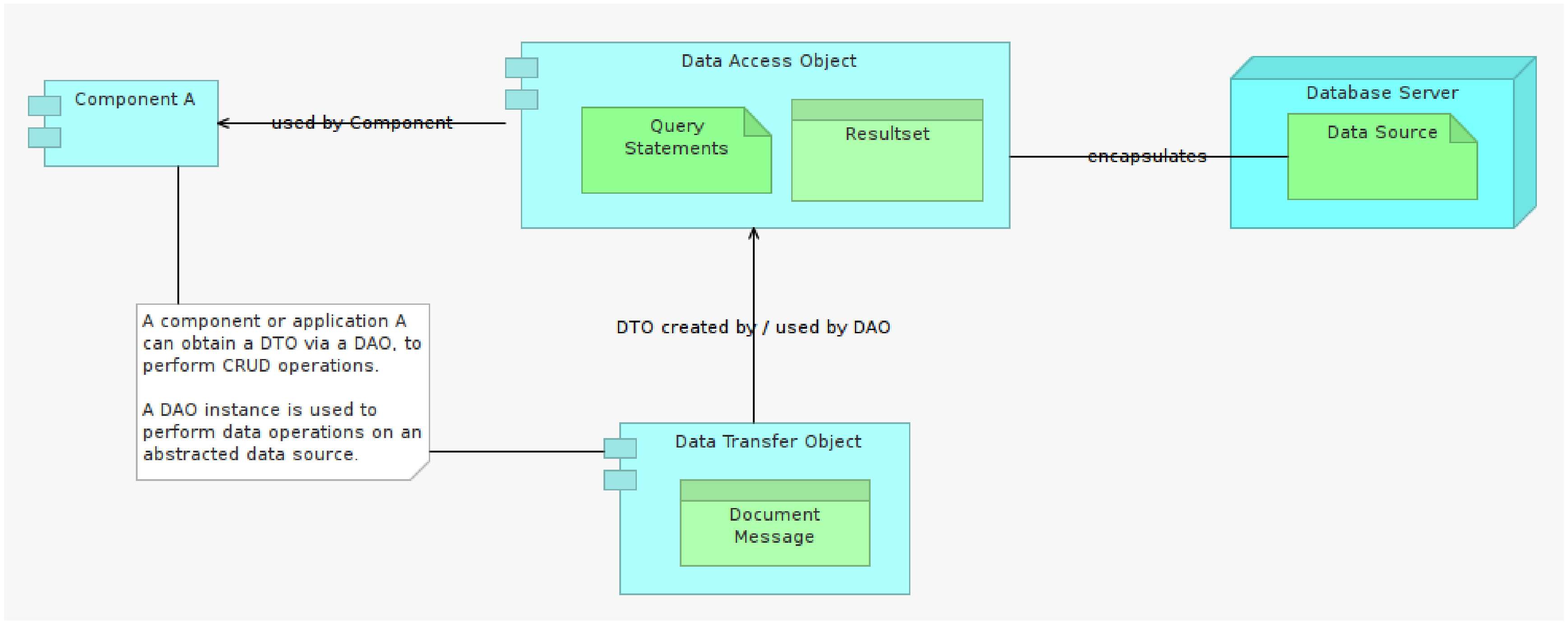
How to abstract business logic from data operations persistence?

Interfaces to data sources vary:

- RDBMS/SQL interfaces can vary
- DBMS can vary
- NoSQL Solutions can vary
- etc.

BY-NC-SA

Overview



A component or application A can obtain a DTO via a DAO, to perform CRUD operations.

A DAO instance is used to perform data operations on an abstracted data source.

(cc) BY-NC-SA

Solution

Use a Data Access Object (DAO) to abstract and encapsulate access to business objects in the data source.

(cc) BY-NC-SA

Example: Table Data Gateway

An object that acts as a Gateway to a database table. One instance handles all the rows in the table.

Mixing SQL in application logic can cause several problems.

Many developers aren't comfortable with SQL, and many who are comfortable may not write it well.

Database administrators need to be able to find SQL easily so they can figure out how to tune and evolve the database.

A Table Data Gateway holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes.

Other code calls its methods for all interaction with the database.

(cc) BY-NC-SA

Example: Active Record

An object that carries both data and behavior.

Much of its data is persistent and needs to be stored in a database.

Active Record uses the most obvious approach, putting data access logic in the domain object.

This way all people know how to read and write their data to and from the database.

(cc) BY-NC-SA

Remarks

Centralizes All Data Access into a central place.

Access to implementation details hidden within DAO.

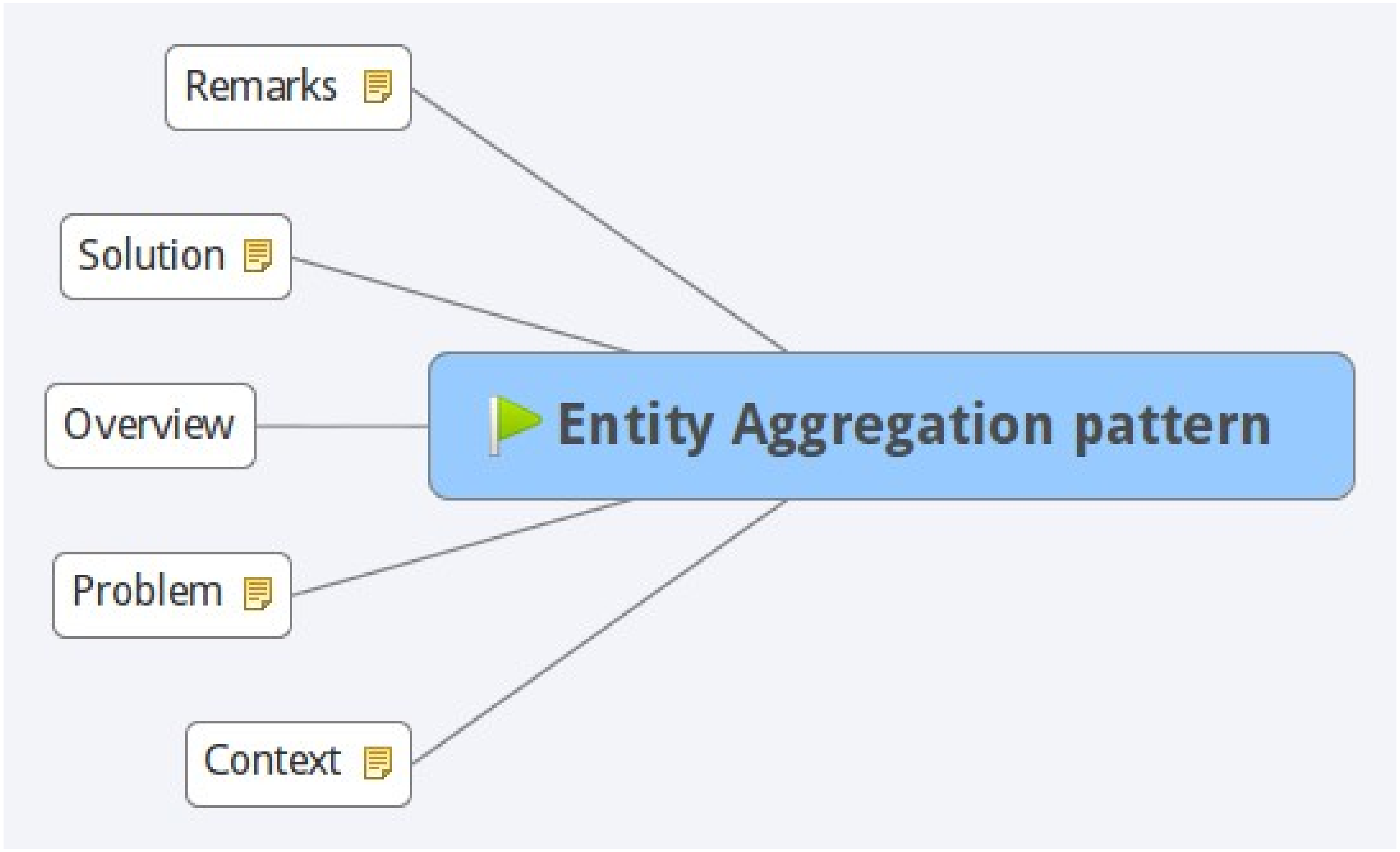
Reduces Code Complexity in Business Logic.

No details, such as SQL, in business logic.

Enables Easier Migration.

(cc) BY-NC-SA

Entity Aggregation pattern



(cc) BY-NC-SA

Context

Enterprise-level data is distributed across multiple repositories in an inconsistent fashion.

Existing applications need to have a single consistent representation of key entities which are logical groups of related data elements such as Customer, Product, Order, or Account.

Moving data between these repositories may not be a viable option.

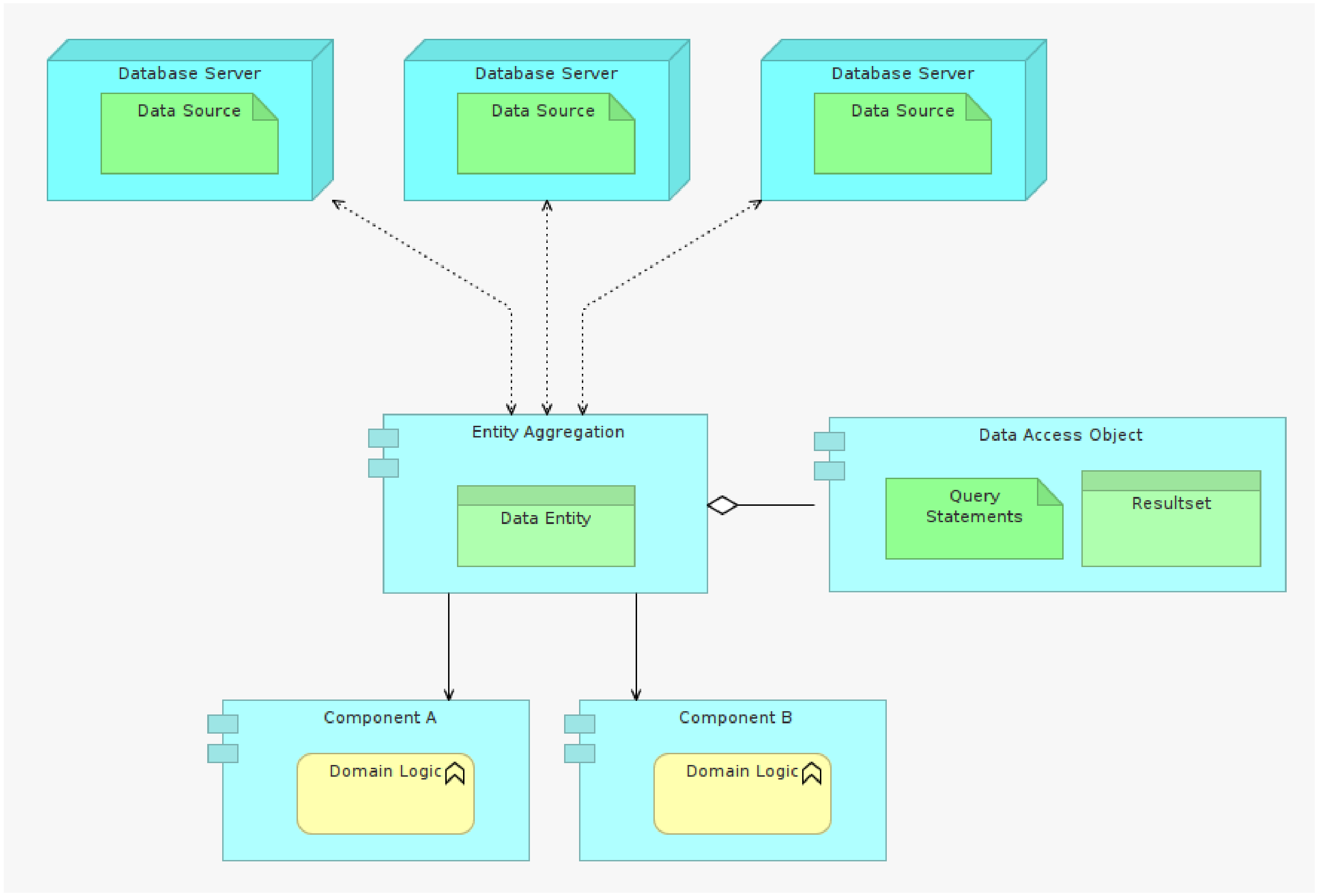
CC BY-NC-SA

Problem

How can enterprise data that is redundantly distributed across multiple repositories be effectively maintained by applications?

BY-NC-SA

Overview



cc BY-NC-SA

Solution

Introduce an Entity Aggregation pattern that provides a logical representation of the entities at an enterprise level with physical connections that support the access and that update to their respective instances in back-end repositories.

cc BY-NC-SA

Remarks

One of the key messages of object orientation is bundling the data with the behavior that uses it.

Two popular variants of the pattern achieve this:

- Domain Model
- Table Module

ORM Frameworks:

- Hibernate

Disadvantages of ORM tools generally stem from the high level of abstraction obscuring what is actually happening in the implementation code.

Also, heavy reliance on ORM software has been cited as a major factor in producing poorly designed databases.

(cc) BY-NC-SA

Domain Model pattern

An object model of the domain that incorporates both behaviour and data.

A Domain Model represents a meaningful individual data record e.g. a single line on an order form.

A Domain Model Component operations expose many different actions to manipulate the record.

(cc) BY-NC-SA

Table Module pattern

A single instance that handles the business logic for all rows in a database table or view.

The primary distinction with Domain Model is that, if you have many orders, a Domain Model will have one order object per order while a Table Module will have one object to handle all orders.

Operations on a Table Module component are about exposing behavior or relationships between records, and perform actions on a set of related records.

BY-NC-SA