Messaging

Infrastrutture per la distribuzione

Claudio Biancalana

Fonti e riferimenti

- Pattern-Oriented Software Architecture (Volume 4): A Pattern Language for Distributed Computing. Wiley, 2007
- Hohpe, G. and Woolf, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004
  - http://www.enterpriseintegrationpatterns.com/
  - http://eaipatterns.com/
- Luca Cabibbo – Roma Tre - ASW

Introduzione alla comunicazione asincrona

- La comunicazione asincrona rappresenta uno stile fondamentale di comunicazione nei sistemi distribuiti
  - È una modalità di comunicazione complementare a quella ‘sincrona’ offerta dall’invocazione remota
  - La comunicazione asincrona è un’astrazione di programmazione distribuita basata sullo scambio di messaggi, che consente ad un componente di inviare un messaggio (contenente **dati** o la **notifica di un evento**), in modo indiretto ed asincrono, ad uno o più componenti, affinché questo o questi componenti possano elaborare il messaggio.
  - Questa modalità di comunicazione è offerta dai **message broker**

Comunicazione asincrona

- La comunicazione asincrona è un paradigma di comunicazione per componenti distribuiti, basato sullo scambio di messaggi, supportato da diversi servizi di middleware, in particolare, dai message broker o più in generale, dal middleware orientato ai messaggi (**message-oriented middleware MOM**)
  - Nella comunicazione asincrona, i componenti distribuiti interagiscono:
    1. inviandosi messaggi (oppure notifiche di eventi)
    2. in modo diretto, con l'ausilio di canali per messaggi
    3. in modo asincrono
  - Per questo, la comunicazione asincrona è anche chiamata messaging

Sulla ‘asincronia’

- Attenzione, non si confonda la **comunicazione asincrona** con **l’invocazione remota asincrona**, infatti, come vedremo tra breve, queste due modalità di comunicazione sono molto diverse tra loro
  - **Nell’invocazione remota asincrona**, il client effettua un’invocazione e poi continua (senza bloccarsi)
    - Tuttavia, al momento dell’invocazione, il client ed il server devono essere entrambi attivi contemporaneamente
  - **Nella comunicazione asincrona,** viceversa, un componente può inviare un messaggio anche quando il componente che riceverà quel messaggio non è attivo o non è nemmeno stato creato, inoltre il componente che riceve il messaggio potrebbe farlo quando il componente che ha inviato il messaggio non è più attivo o è stato distrutto

Produttori, messaggi, consumatori

- Nella comunicazione asincrona, ciascuna singola interazione ha lo scopo di consentire ad un componente ‘produttore’ di trasmettere dati (un ‘messaggio’) ad un altro componente (‘consumatore’).
  - Un primo componente (nel ruolo di produttore) ha dei dati che vuole che vengano elaborati, ma lui non può elaborarli direttamente
    - I dati potrebbero essere dei dati strutturati, un documento, una richiesta per un servizio (un comando) o una risposta, oppure la notifica di un evento
  - Allora, il produttore prepara un messaggio contenente questi dati, e poi lo invia ad un altro componente, che è in grado di elaborare oppure è interessato a elaborare quei dati
  - Quindi, un secondo componente (nel ruolo di consumatore) riceve questo messaggio e poi ne estrae i dati e li elabora

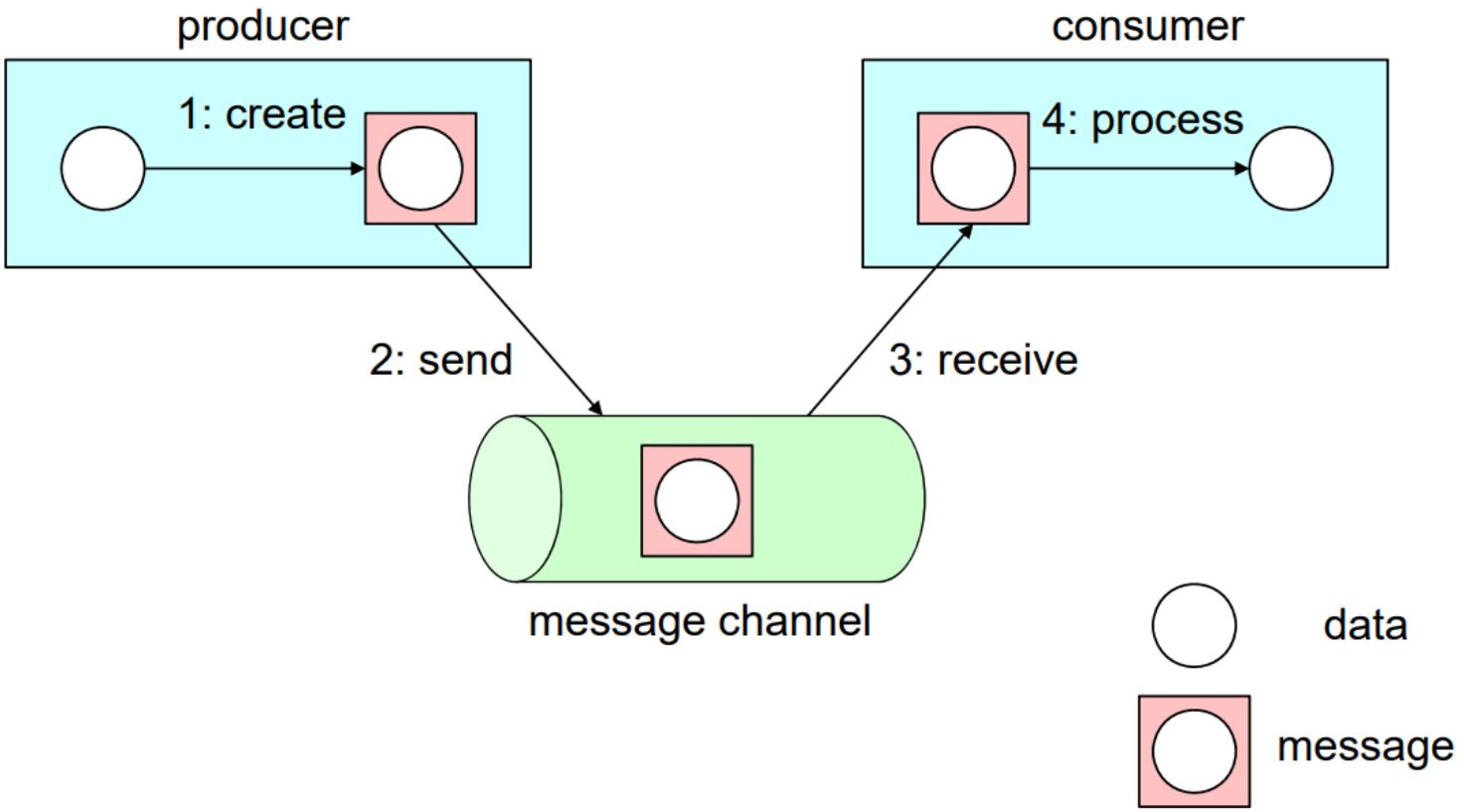
La comunicazione è indiretta

- La comunicazione è **indiretta**, infatti, il componente produttore non trasmette il messaggio direttamente ad uno specifico componente consumatore
  - Piuttosto, il produttore invia il messaggio ad **un canale per messaggi**, specifico per il tipo del messaggio trasmesso, ma senza specificare l'identità del consumatore del messaggio
    - Infatti, il produttore in genere non sa (o comunque non è interessato a sapere) qual è il componente che riceverà ed elaborerà il suo messaggio
  - Quindi, un consumatore riceve il messaggio prelevandolo da quel canale per messaggi
    - Il consumatore viene scelto tra quelli registrati presso quel canale per messaggi, ovvero, tra i componenti che sono in grado di elaborare quello specifico tipo di messaggi
  - Dunque, il canale per i messaggi costituisce **un'indirezione spaziale** nella comunicazione tra produttore e consumatore

La comunicazione è asincrona

- La comunicazione è **asincrona** nel senso che l'invio e la ricezione del messaggio non sono sincronizzate
  - Dopo che il produttore ha inviato il suo messaggio al canale per messaggi, prosegue nel suo lavoro
    - Il produttore non attende né che il messaggio sia stato ricevuto da un consumatore né che sia stato elaborato
  - Il consumatore del messaggio riceve ed elabora il messaggio appena è disponibile a farlo
    - Questo potrebbe avvenire subito dopo che il messaggio è stato inviato nel canale per messaggi, ma anche piu tardi!
    - Inoltre il consumatore in genere non comunica con li produttore del messaggio ricevuto né per notificargli la ricezione del messaggio tantomeno la sua elaborazione
  - Dunque, il canale per messaggi costituisce anche **un'indirezione temporale** nella comunicazione tra produttore e consumatore

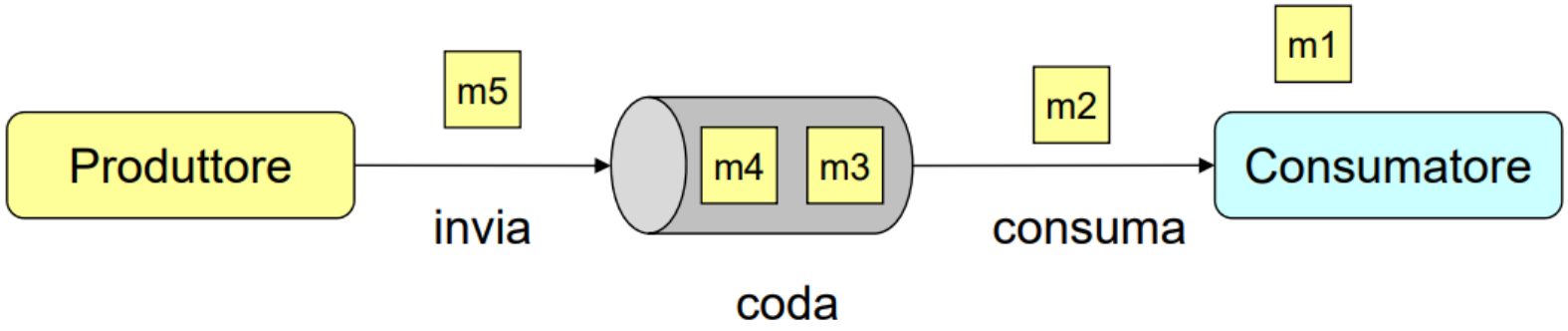
Comunicazione indiretta ed asincrona



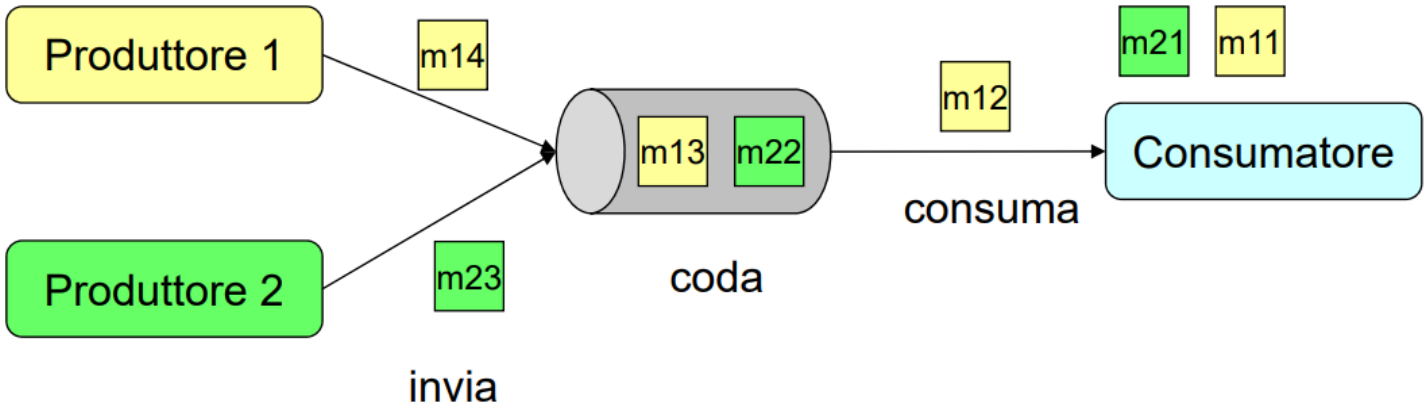
Invocazione implicita

- Quando un consumatore riceve un messaggio, di solito lo elabora eseguendo un'operazione o un metodo opportuno
  - L'operazione da eseguire viene scelta dal consumatore del messaggio, sulla base del contenuto del messaggio
    - L'operazione da eseguire non viene scelta sulla base di una richiesta diretta da parte del produttore del messaggio (che di solito non sa nemmeno qual è l'interfaccia procedurale del consumatore)
  - **Per questo si parla anche di invocazione implicita**
    - Infatti, l'invocazione dell'operazione eseguita per elaborare il messaggio è, appunto, 'implicita'

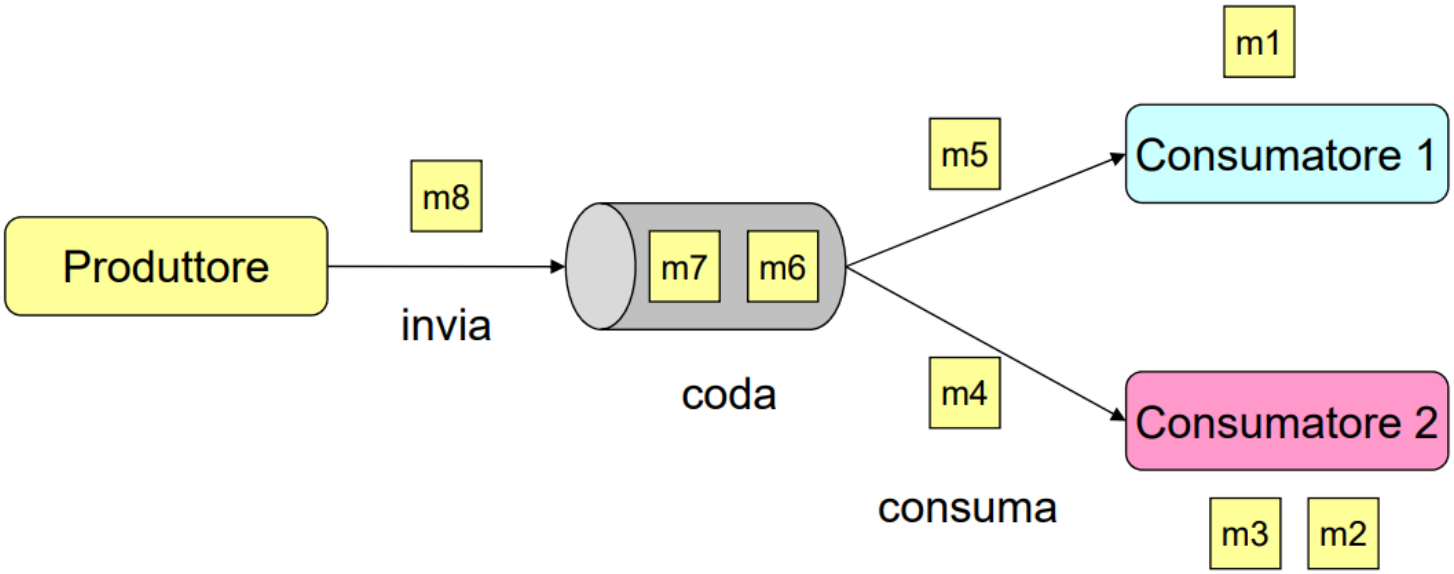
Canale point-to-point



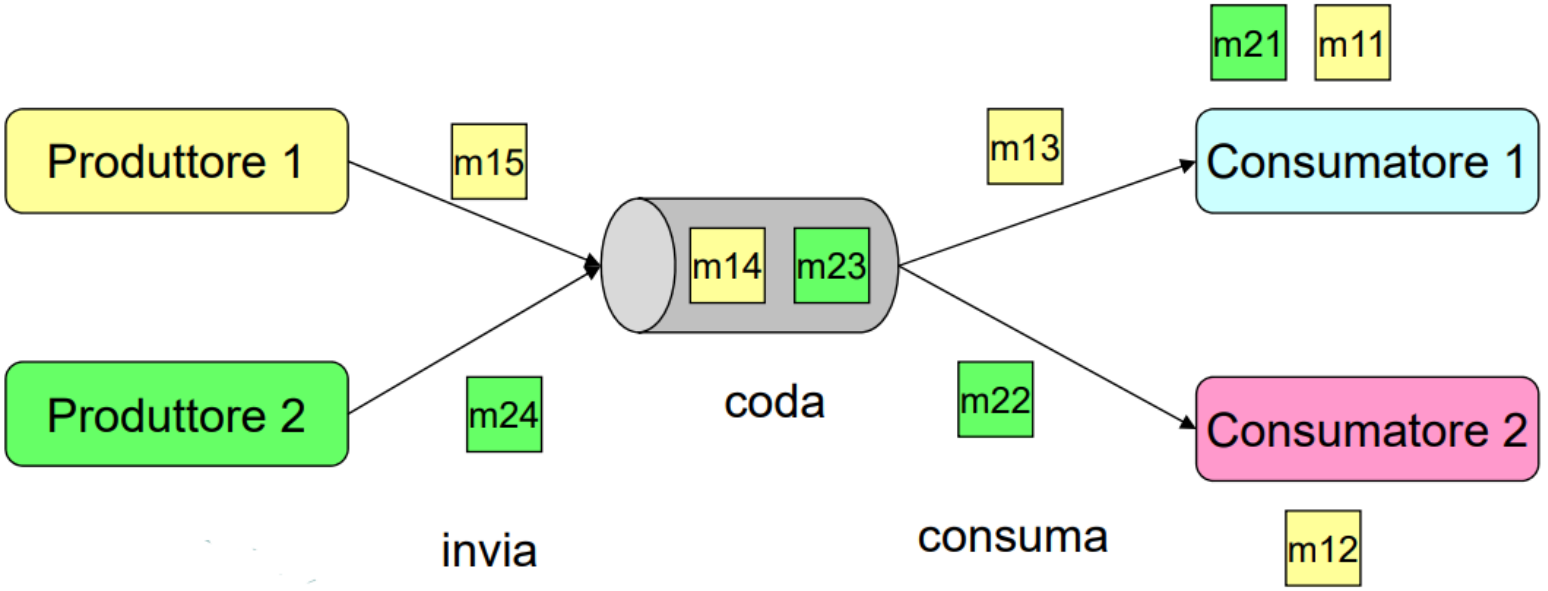
Canale point-to-point: più produttori



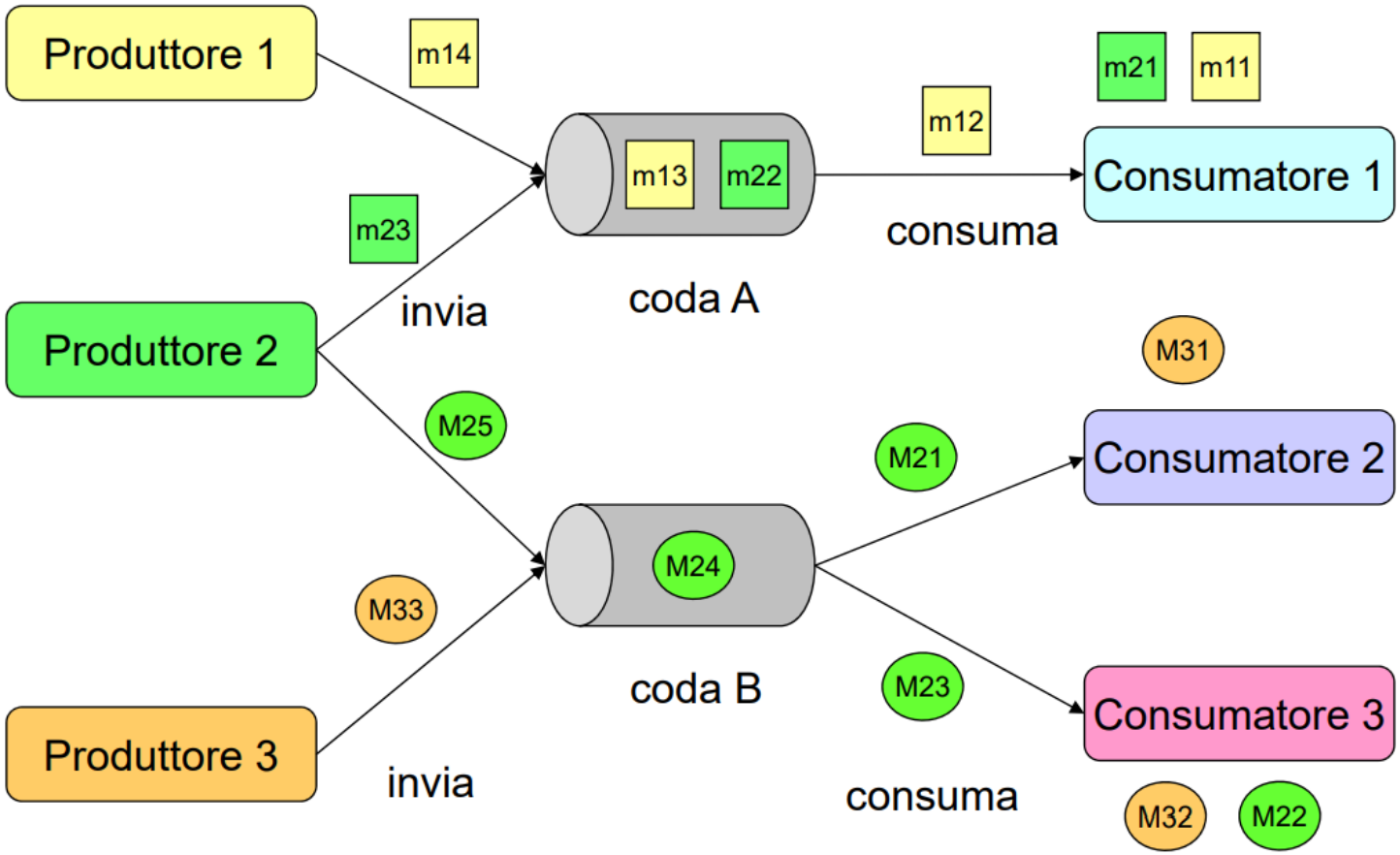
Canale point-to-point: più consumatori



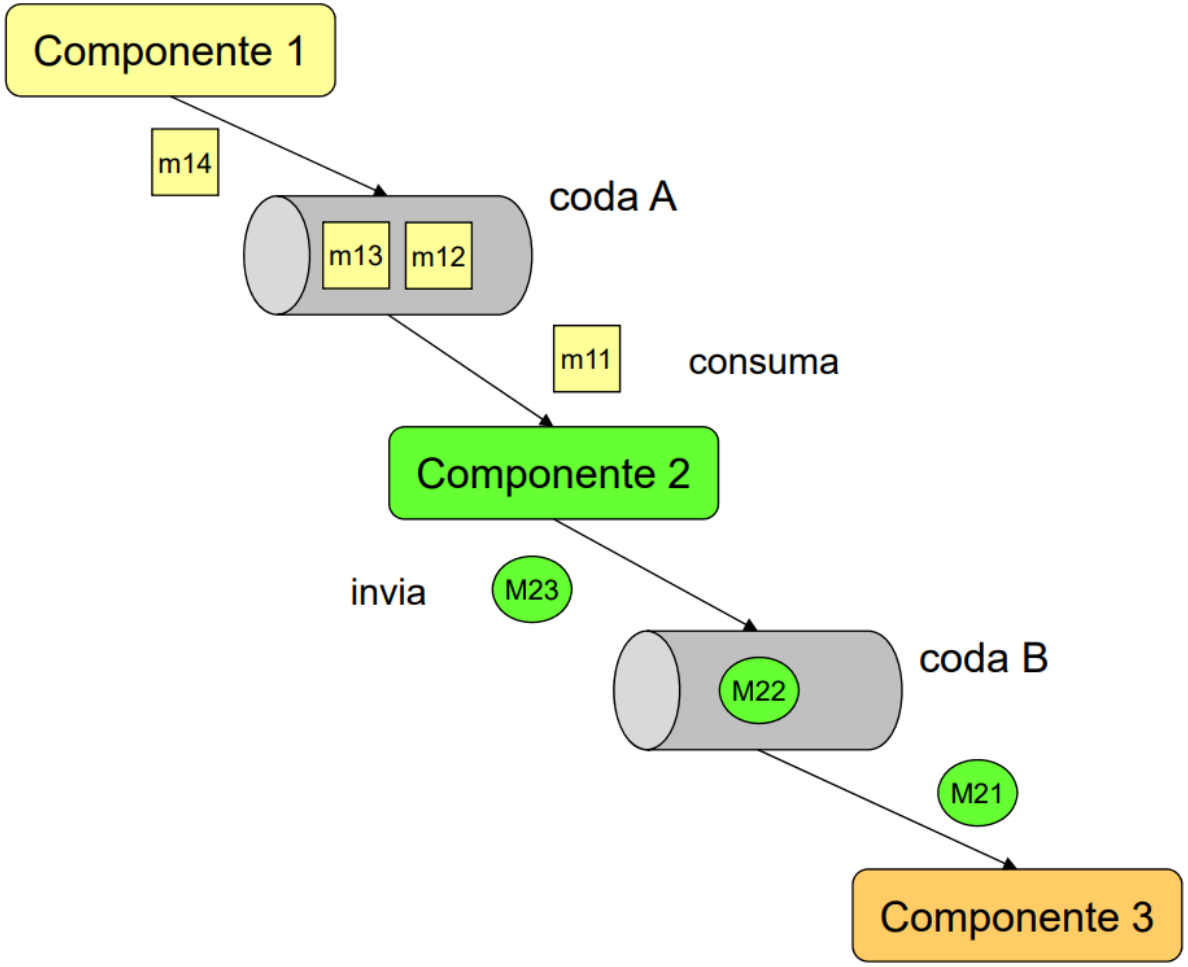
Canale point-to-point: più produttori e più consumatori



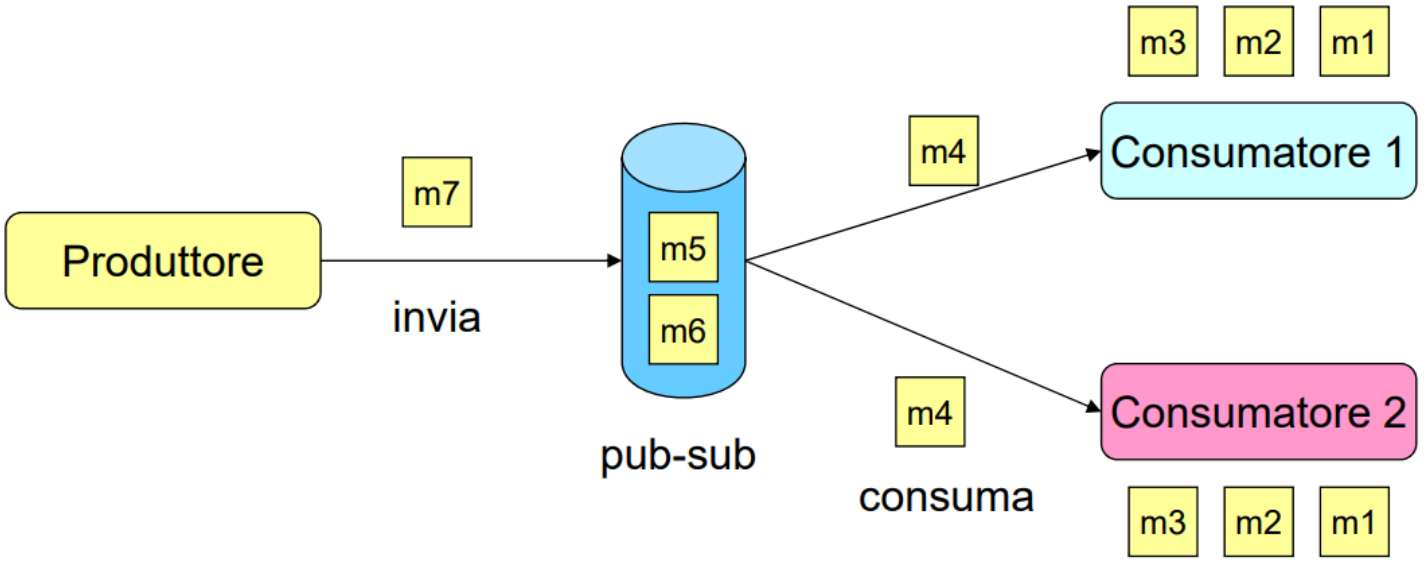
Canale point-to-point: più canali (più tipi di messaggi)



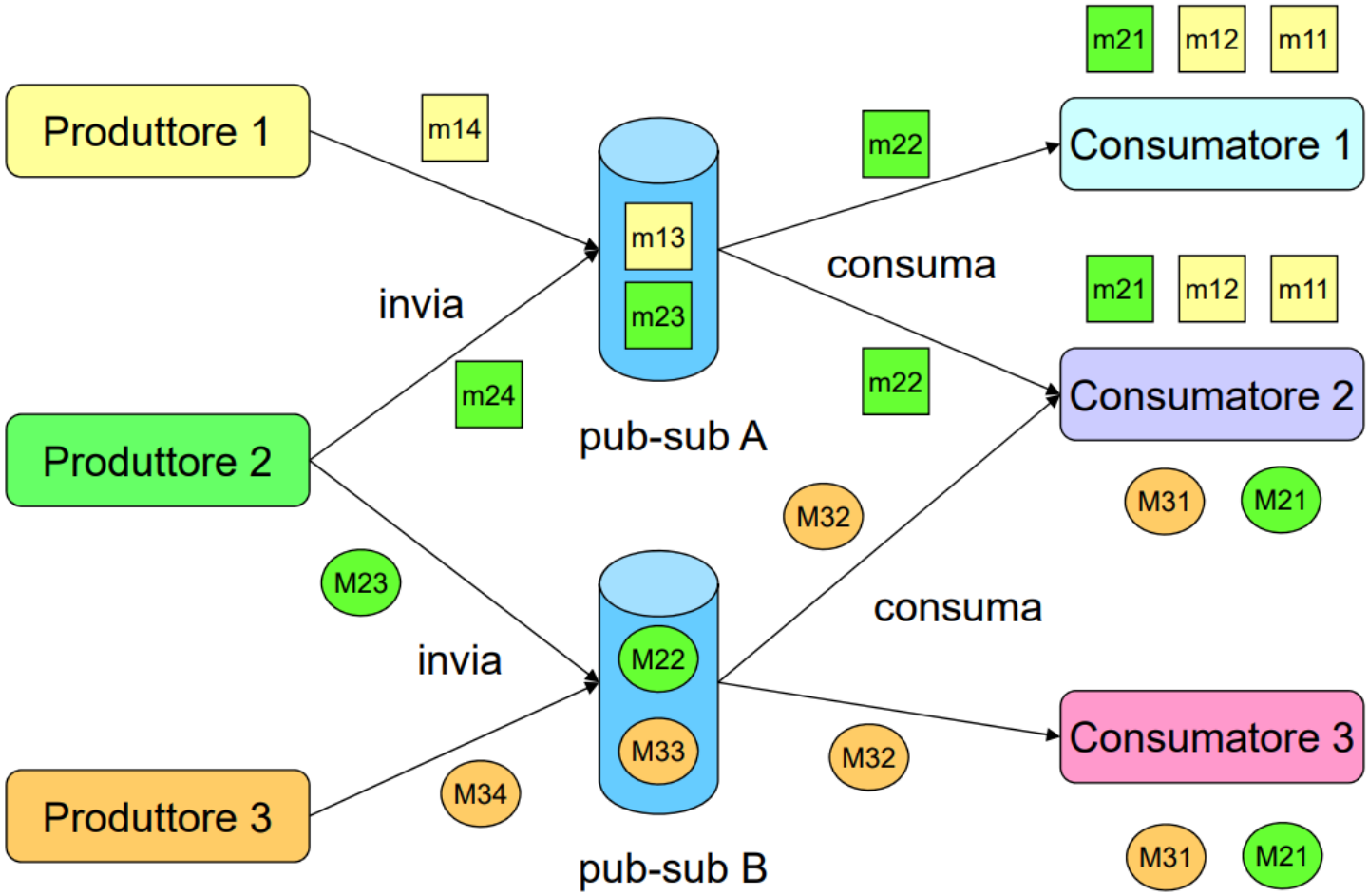
Canale point-to-point: componenti consumatori/produttori



Canale publish-subscribe



Canale publish-subscribe



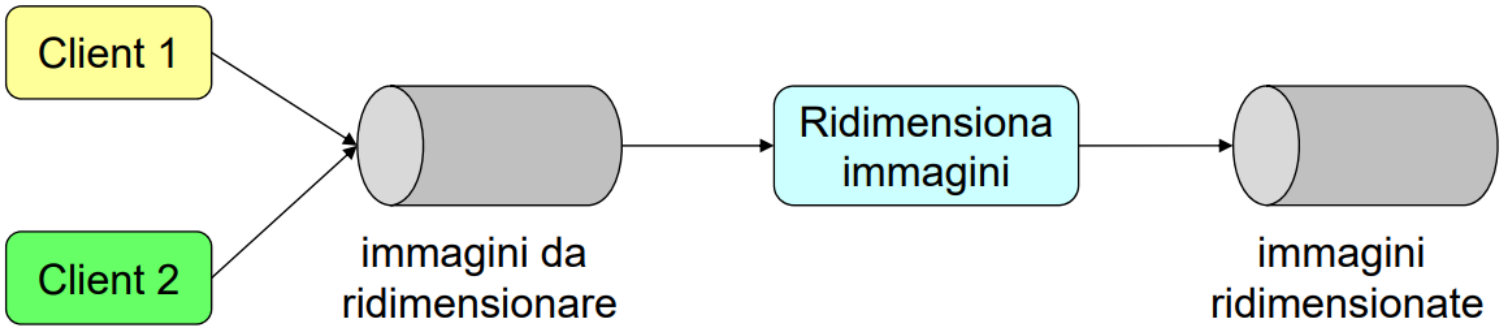
Esempio (1)

- Prima di andare avanti, è utile un esempio per descrivere un possibile scenario di applicazione della comunicazione asincrona
  - Consideriamo dei componenti che hanno dei dati che devono essere elaborati
    - Ad esempio, componenti che ricevono messaggi da elaborare dagli utenti di un'applicazione
  - Per ciascuno dei dati va eseguito un certo compito
    - Ad esempio, un ridimensionamento dell'immagine
  - Il compito si può eseguire indipendentemente su ciascun dato
    - Ad esempio, separatamente, per ciascuna immagine

Esempio (2)

- C’è un componente in grado di svolgere questo compito
- Il compito non va necessariamente svolto in modo sincrono, sulla base di un protocollo di richiesta risposta
- Ciascun compito può produrre dei risultati, ma non necessariamente una risposta al componente che l’ha richiesto

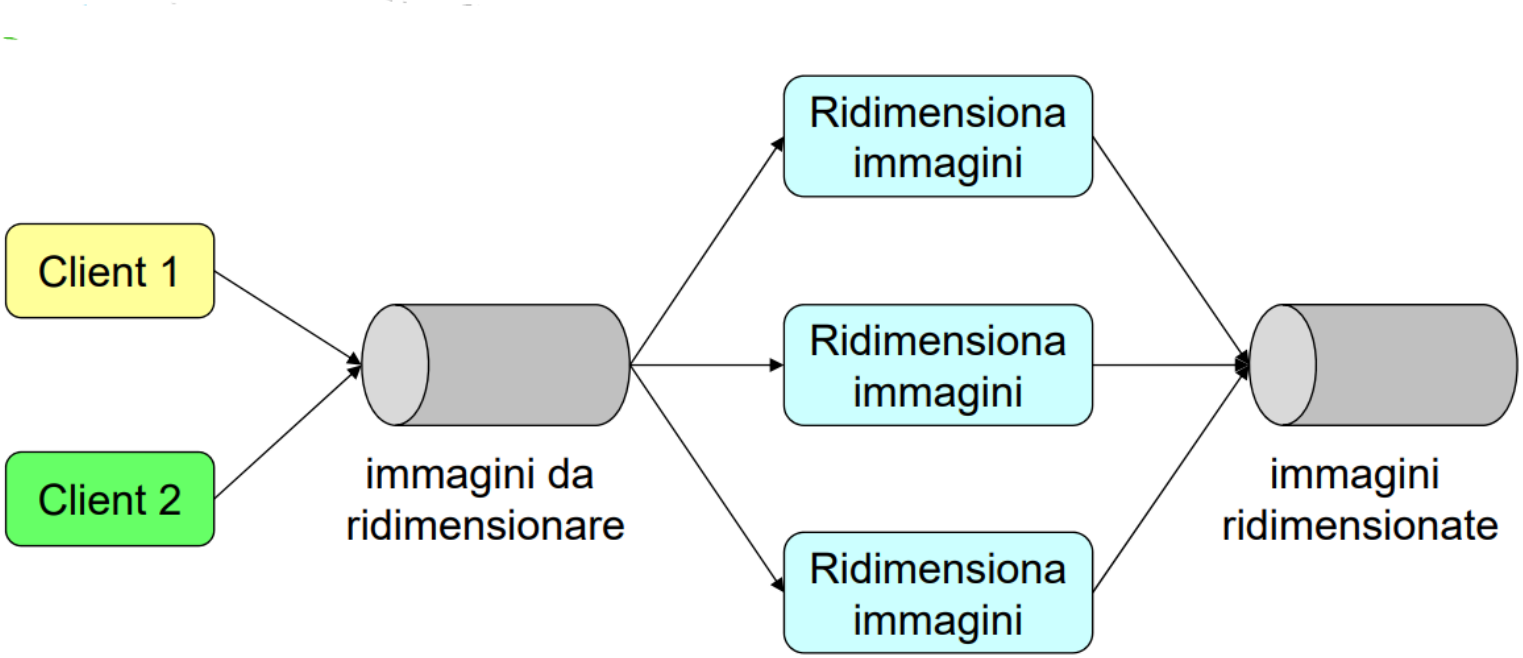
Soluzione


Client 1
Client 2
immagini da ridimensionare
Ridimensiona immagini
immagini ridimensionate

Discussione

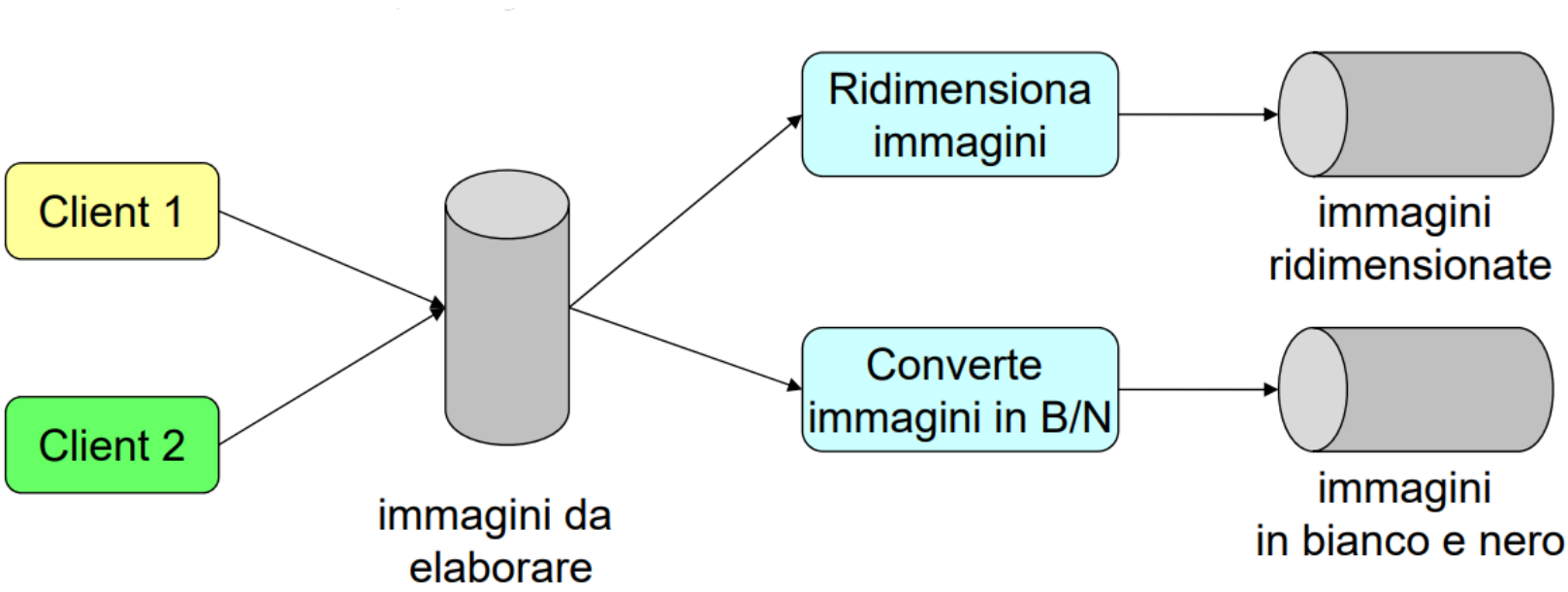
- In questo caso, l'uso di un canale point-to-point consente
  - Ai componenti produttori di inviare messaggi per chiedere l'elaborazione dei loro dati anche quando attualemente non ci sono componenti consumatori pronti a svolgere il compito richiesto, potrebbero essere attivati successivamente
  - Di avere più istanze di componenti consumatori in grado di svolgere il compito richiesto, in modo da poter distribuire il carico dei compiti sulle diverse istanze presenti
  - Di poter variare dinamicamente il numero di istanze di componenti consumatori per svolgere il compito richiesto in caso di aumento del carico dei messaggi da elaborare, e di poterlo diminuire quando il carico diminuisce, sostenendo così la scalabilità nell'esecuzione del compito

Soluzione


Client 1
Client 2
immagini da ridimensionare
Ridimensiona immagini
Ridimensiona immagini
Ridimensiona immagini
immagini ridimensionate

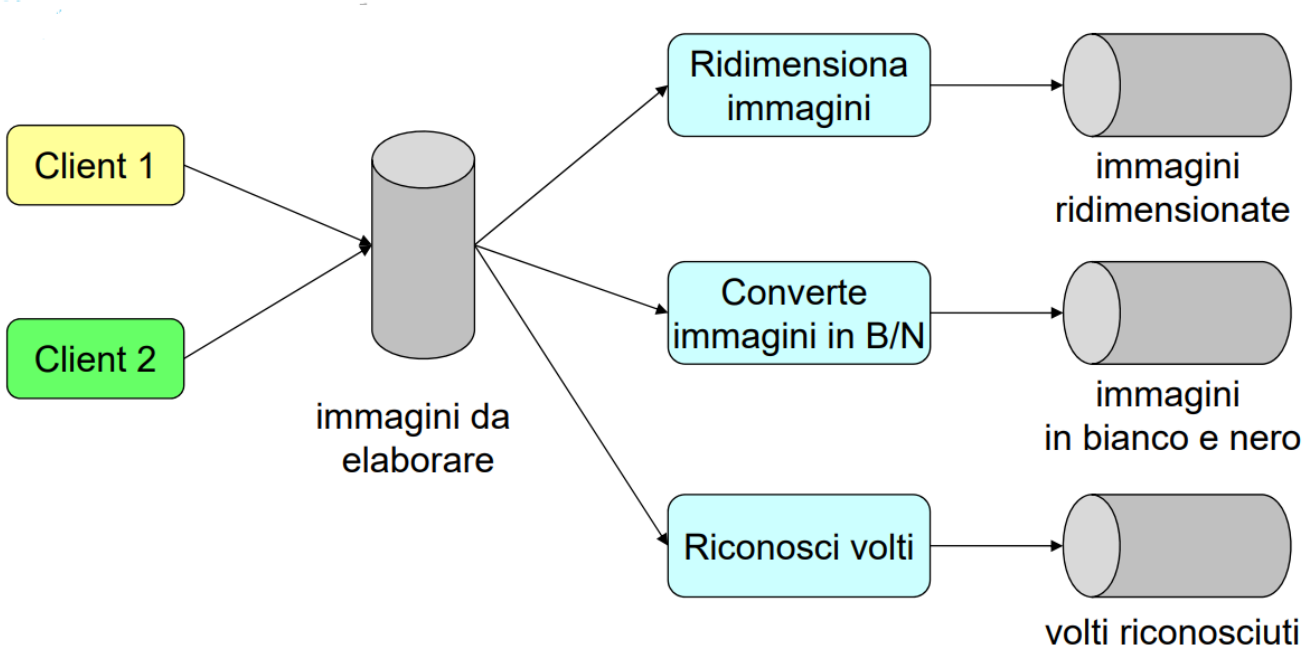
Discussione

- L’uso di un canale publish-subscribe è utile quando per ciascun messaggio vanno svolti più compiti separati



Discussione

- Un canale di publish-subsribe consente di aggiungere dinamicamente nuovi componenti per svolgere dei compiti aggiuntivi



Discussione

- Questo esempio consente di comprendere il supporto fornito dalla comunicazione asincrona a diverse qualità
  - La modificabilità in virtù dell'accoppiamento basso tra componenti
  - Prestazione e scalabilità, anche grazie alla possibilità di replicare componenti

Esempio
Esplorazione Grafo

- Nella visita generica di un grafo si fa uso di un insieme F, o frangia, che contiene almeno tutti i nodi non ancora visitati ma adiacenti a nodi visitati.
- Poiché un nodo in un grafo può essere in generale adiacente a nodi diversi, quando si inserisce un nodo in F conviene controllare che non sia già presente (perché inserito precedentemente).

- A seconda della struttura-dati impiegata per F, si ottengono, come nel caso degli alberi, i diversi tipi di visita:
  - coda: visita in ampiezza
  - coda con priorità: esempio A*
  - pila: visita in profondità

Approfondimenti

Premessa

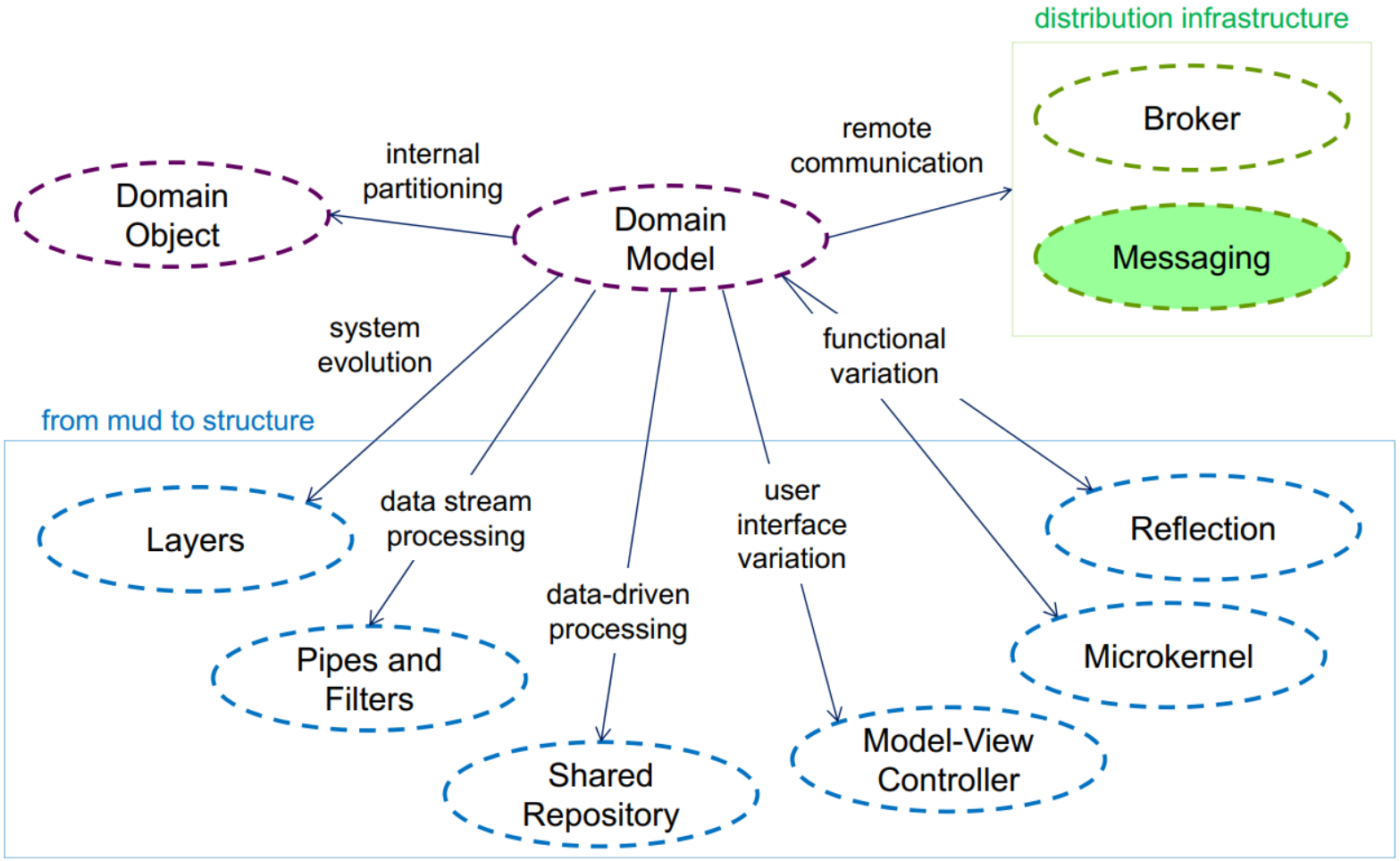
- In questa seconda parte viene presentata la coppia di pattern architetturali **Messaging** e **Publisher-Subscriber**, che supportano lo stile di comunicazione asincrona
  - Questi due pattern hanno l'obiettivo di favorire e semplificare l'utilizzo della comunicazione asincrona
- Tra questi due pattern ci sono molti punti in comune, ma anche alcune differenze
  - In un modo un po' improprio, il nome usato per questo pattern unificato è Messaging

Introduzione

Messaging è un pattern architetturale della categoria “infrastrutture per la distribuzione”

- questo pattern supporta lo stile di comunicazione della comunicazione asincrona
- concettualmente, ci sono delle importanti differenze tra i pattern Broker e Messaging
- il pattern Broker descrive l’implementazione, nel middleware, dell’infrastruttura di comunicazione che abilita l’invocazione remota
- il pattern Messaging si occupa invece soprattutto degli obiettivi e dell’applicazione della comunicazione asincrona – ovvero, della progettazione dell’infrastruttura di comunicazione (da realizzare usando un message broker) dei sistemi software basati sulla comunicazione asincrona

Stili architetturali
Pattern Oriented Software Atchitecture (POSA)



Messaging

Messaging organizza un sistema distribuito come un insieme di componenti che interagiscono tramite lo scambio di messaggi

- i messaggi possono contenere **dati**, **informazioni**, **metadati**, ma anche **richieste**, **risposte** e **informazioni di errore**, oppure anche **notifiche di eventi**
- lo **scambio** di messaggi avviene in modo **indiretto e asincrono**, tramite un insieme di canali per messaggi
- i componenti che producono messaggi possono comunicare con i componenti che consumano messaggi in modalità molti-a-uno (*point-to-point*) oppure molti-a-molti (*publish-subscribe*)
- questo stile di comunicazione **rilassa accoppiamento e tipizzazione**
- i messaggi scambiati devono essere tipizzati – ma i componenti non sono accoppiati tramite operazioni e interfacce fortemente tipizzate
- è **rilassato anche l'accoppiamento spaziale e temporale**

Relazione con la comunicazione asincrona

- Il pattern architetturale Messaging sostiene e favorisce la comunicazione asincrona – basata sullo scambio di messaggi e sulla notifica di eventi – cercando di semplificare il suo utilizzo
  - per affrontare la complessità e le sfide poste agli sviluppatori dal paradigma di programmazione distribuita basato sulla comunicazione asincrona e sullo scambio di messaggi
  - per perseguire le qualità sostenute da questa modalità di comunicazione – come modificabilità (accoppiamento basso e maggior flessibilità nella comunicazione), prestazioni, scalabilità, affidabilità e disponibilità

Pattern Messaging

- Il pattern architetturale Messaging
  - nella categoria “infrastrutture per la distribuzione”
  - definisce un’infrastruttura di comunicazione, per sostenere l’integrazione di componenti sviluppati in modo indipendente (autonomo) in un sistema coerente
  - la comunicazione è basata sullo scambio di messaggi (o documenti o eventi) tra i diversi componenti

Pattern Messaging
Contesto

- Integrazione di un insieme di componenti (o servizi) sviluppati in modo indipendente
- Ad esempio, si vogliono comporre i componenti di un insieme di applicazioni esistenti, per realizzare una nuova applicazione, con un maggior valore di business

Messaging
Problema

- Alcuni sistemi distribuiti sono composti da componenti (o servizi) sviluppati indipendentemente tra loro
- Questi componenti devono essere integrati, in modo da formare un sistema coerente
- E' dunque necessario far interagire e collaborare questi componenti, anche se i componenti sono stati sviluppati indipendentemente, e dunque non sono di solito a conoscenza l'uno dell'altro
- Interazione tra componenti
  - **Deve essere basata su un accoppiamento debole**
  - **Deve essere flessibile**
  - **Deve avvenire in modo affidabile**

Messaging
Soluzione

- **Collega i componenti (o servizi) mediante un bus per messaggi**, che consente ai componenti di scambiarsi messaggi in modo asincrono
  - Realizza il collegamento tra componenti (o servizi) di interesse anche mediante l'ausilio di ulteriori elementi software
- **Codifica i messaggi in modo che i loro mittenti (produttori) e destinatari (consumatori) possano comunicare in modo affidabile**, e senza dover conoscere staticamente tutte le informazioni sui tipi di dati
  - I messaggi possono incapsulare informazioni, strutture dati, richieste e/o risposte, oppure notifiche di eventi
  - Inoltre, i messaggi sono di solito auto-descrittivi

Discussione (1)

- La soluzione proposta dal pattern architetturale **Messaging** al problema dell'integrazione di componenti o servizi indipendenti richiede certamente dei chiarimenti
  - un'altra idea centrale della soluzione è di realizzare le interazioni tra i componenti di interesse (possibilmente per tramite degli elementi "collante") sulla base dello **scambio asincrono di messaggi, tramite un bus per messaggi** (ovvero, un insieme di canali per messaggi) – dunque, **mediante la comunicazione asincrona** (e non invocazioni remote o altro)
  - la comunicazione asincrona sostiene infatti un accoppiamento debole – che può abilitare l'integrazione di componenti e servizi
  - anche i messaggi e i canali per messaggi possono essere progettati applicando i pattern di supporto al pattern architetturale Messaging (descritti nel seguito)

Discussione (2)

- quando un componente riceve un messaggio, in genere si occupa dell'elaborazione delle informazioni contenute nel messaggio – questo può portare allo **scambio di ulteriori messaggi**
- complessivamente, i diversi componenti possono usare i messaggi per guidare o coordinare le loro computazioni
- i messaggi possono anche codificare la notifica di eventi – che devono essere propagati da componenti «publisher» a componenti «subscriber»

Pros

- **Benefici**
  - **Modificabilità e flessibilità**, è possibile aggiungere/rimuovere/sostituire componenti
  - **Affidabilità**, possibile la consegna affidabile di messaggi
  - **Prestazioni**, la comunicazione asincrona può favorire la concorrenza, inoltre è possibile la replicazione dei consumatori di messaggi

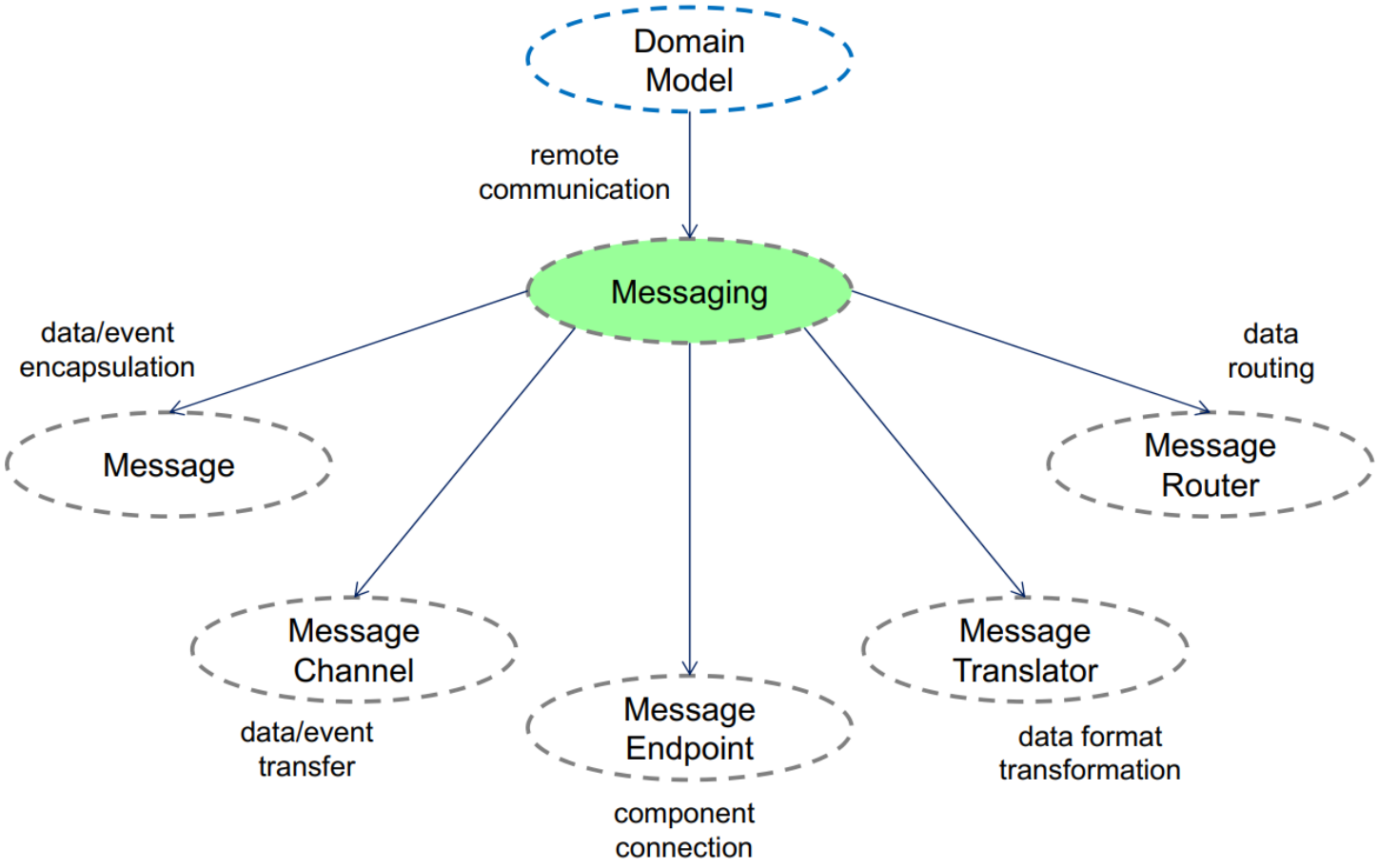
Cons

- **Svantaggi**
  - **Complessità**, maggior complessità del paradigma di programmazione distribuita basato sullo scambio asincrono di messaggi
  - **Prestazioni**, overhead dovuto alla gestione dei canali per messaggi e dei messaggi
  - **Prestazioni**, overhead dovuto alla necessità di codificare/decodificare i messaggi
  - **Verifica e affidabilità**, la mancanza di interfacce tipizzate staticamente (o della loro conoscenza) può rendere difficile verificare/validare il comportamento del sistema

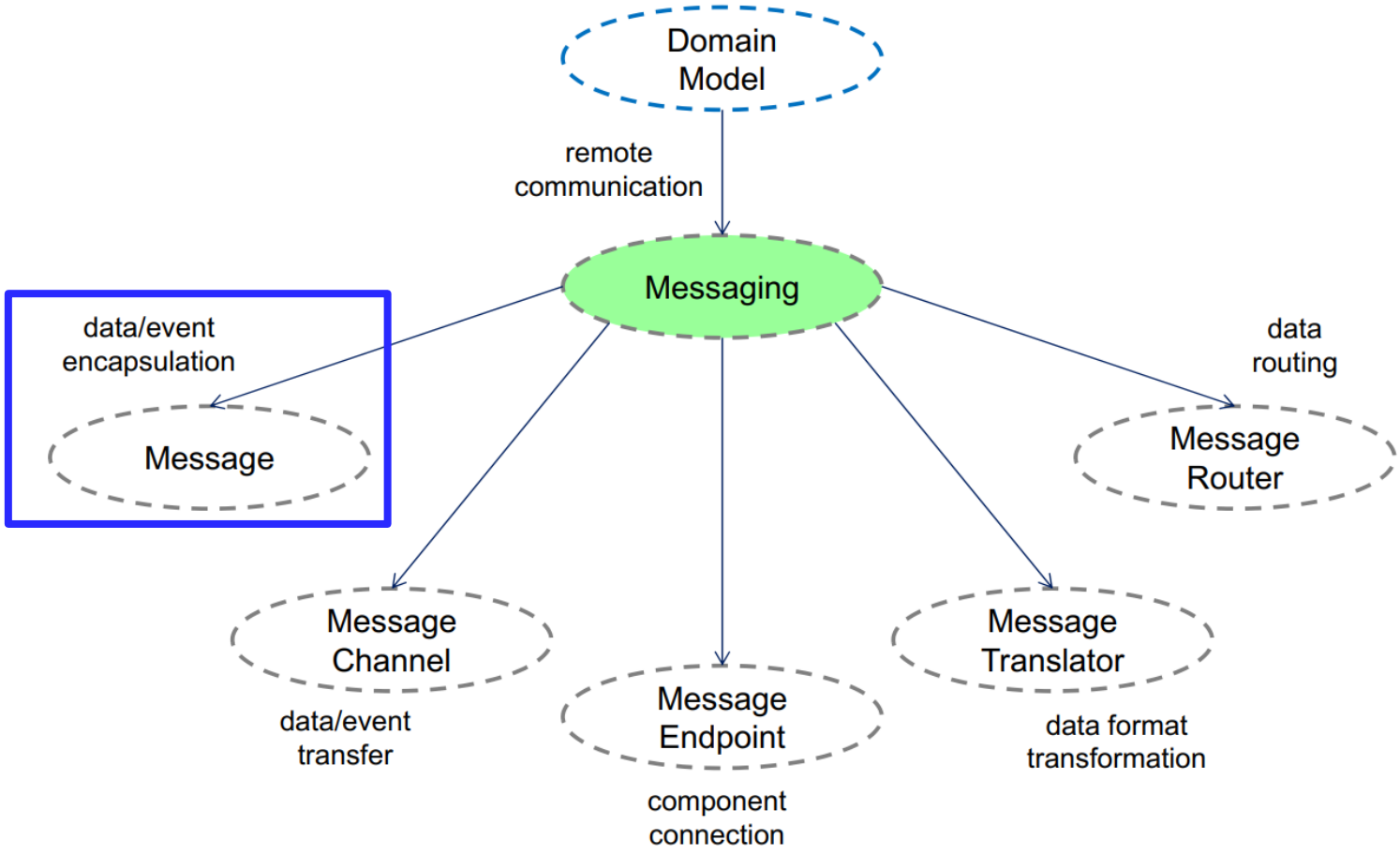
Ulteriori pattern di supporto

- L'applicazione del pattern architetturale Messaging è favorita dall'utilizzo di ulteriori pattern architetturali, più specifici, che affrontano in modo coerente alcuni problemi ricorrenti della comunicazione asincrona – tra cui
  - la comunicazione avviene mediante lo **scambio di messaggi**
  - la comunicazione avviene tramite **canali per messaggi**
  - i componenti vengono collegati ai canali mediante degli elementi detti **message endpoint** – alcuni message endpoint sono realizzati nella forma di adattatori
  - è anche possibile avere degli **ulteriori componenti aggiuntivi** ad es., elementi che si occupano della trasformazione di messaggi ("filtri") oppure del routing di messaggi
  - questi pattern hanno, tra l'altro, lo scopo di mantenere basso l'accoppiamento necessario tra i componenti (solitamente indipendenti) che hanno tuttavia necessità di comunicare

Ulteriori pattern di supporto



Ulteriori pattern di supporto



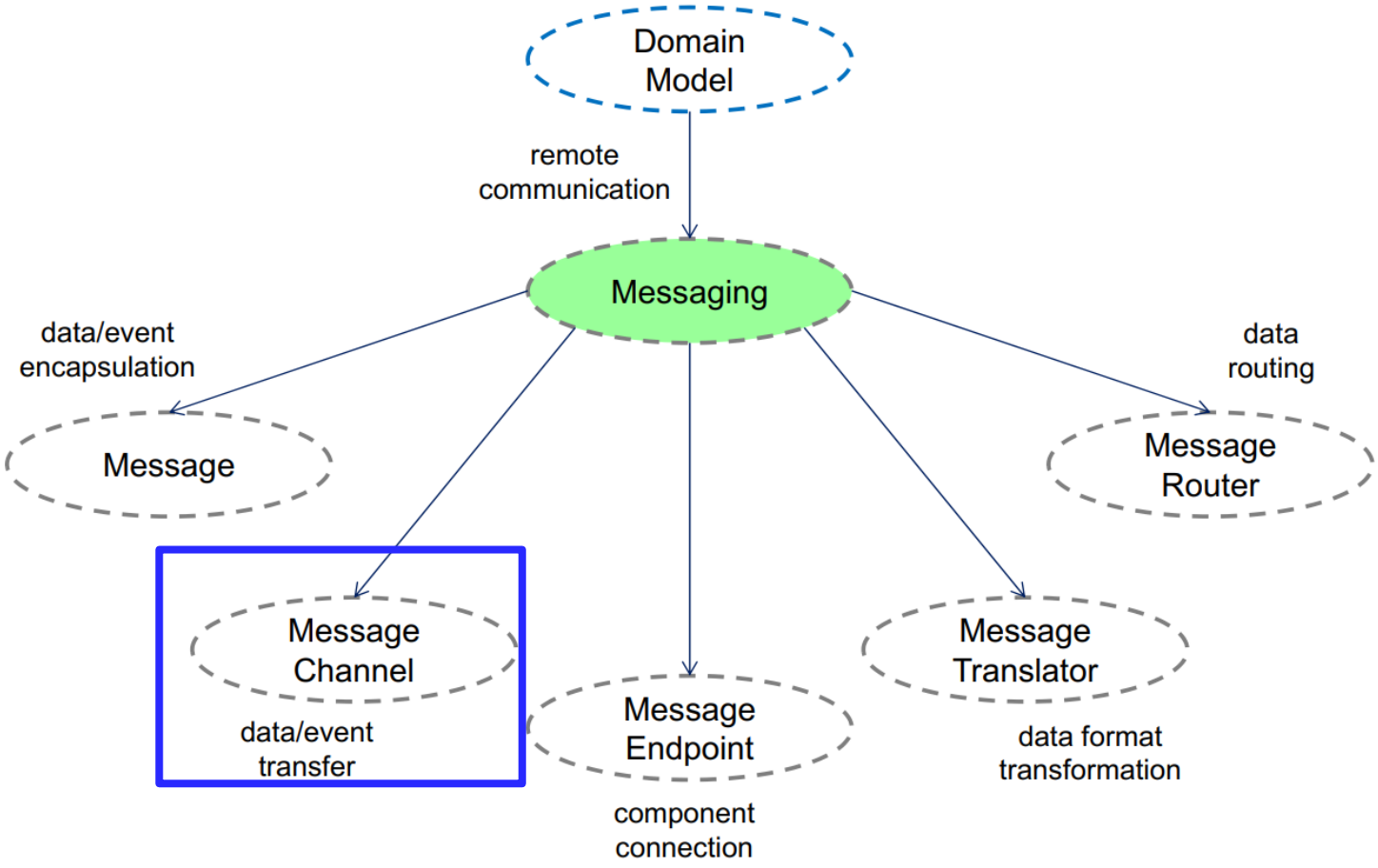
Message

- Problema
  - Come è possibile connettere dei componenti o servizi per consentire lo scambio flessibile di informazioni?
- Soluzioni
  - incapsula le informazioni da scambiare (come dei dati, un documento, la notifica di un evento oppure anche la richiesta di invocazione di un'operazione o di servizio o una risposta) in un messaggio (**message**)
  - il messaggio è formato da
    - un body (o payload) – contiene i dati effettivi
    - un header – specifica ulteriori metadati circa i dati trasmessi ad es., origine, destinazione, dimensione, scadenza, correlation id, ...

Message

- Conseguenze
  - consente il trasferimento di strutture di dati, documenti ed eventi tra componenti
    - i dati posseduti da un componente possono essere codificati in un messaggio, trasmessi ad altri componenti, e da questi ricostruiti
  - sostiene un accoppiamento debole
    - conoscere il formato dei messaggi accettati da un altro componente (destinatario del messaggio) è una forma di accoppiamento più debole che non conoscere l'interfaccia procedurale (staticamente tipata) dell'altro componente
- flessibilità
  - se i messaggi sono autodescrittivi – ad es., usando un formato di interscambio come JSON o XML
- overhead nella codifica/decodifica dei messaggi

Ulteriori pattern di supporto



Message Channel

- Problema
  - i messaggi contengono solo i dati che devono essere scambiati tra componenti o servizi
  - per sostenere un accoppiamento debole, i componenti o servizi non dovrebbero conoscere chi è interessato a quali messaggi
  - è comunque necessario un meccanismo per connettere i componenti o servizi, per consentire lo scambio di messaggi

Message Channel

- Soluzione
  - non connettere direttamente i componenti che devono interagire – piuttosto, collegali tramite un canale per messaggi (**message channel**) che gli consenta di scambiare messaggi
  - quando un componente produttore deve comunicare un messaggio, lo inoltra ad un canale per messaggi
  - i componenti consumatori interessati al messaggio (oppure che sono disponibili ad elaborarlo) lo possono poi prelevare dal canale ed elaborare

Message Channel

- Soluzione
  - Un canale per messaggi è dunque un «indirizzo logico» presso i cui componenti e servizi possono inviare e ricevere messaggi
  - In un sistema basato sulla comunicazione asincrona è comune l’uso di una molteplicità di canali per messaggi, ciascuno relativo ad una specifica tipologia dimessaggi.

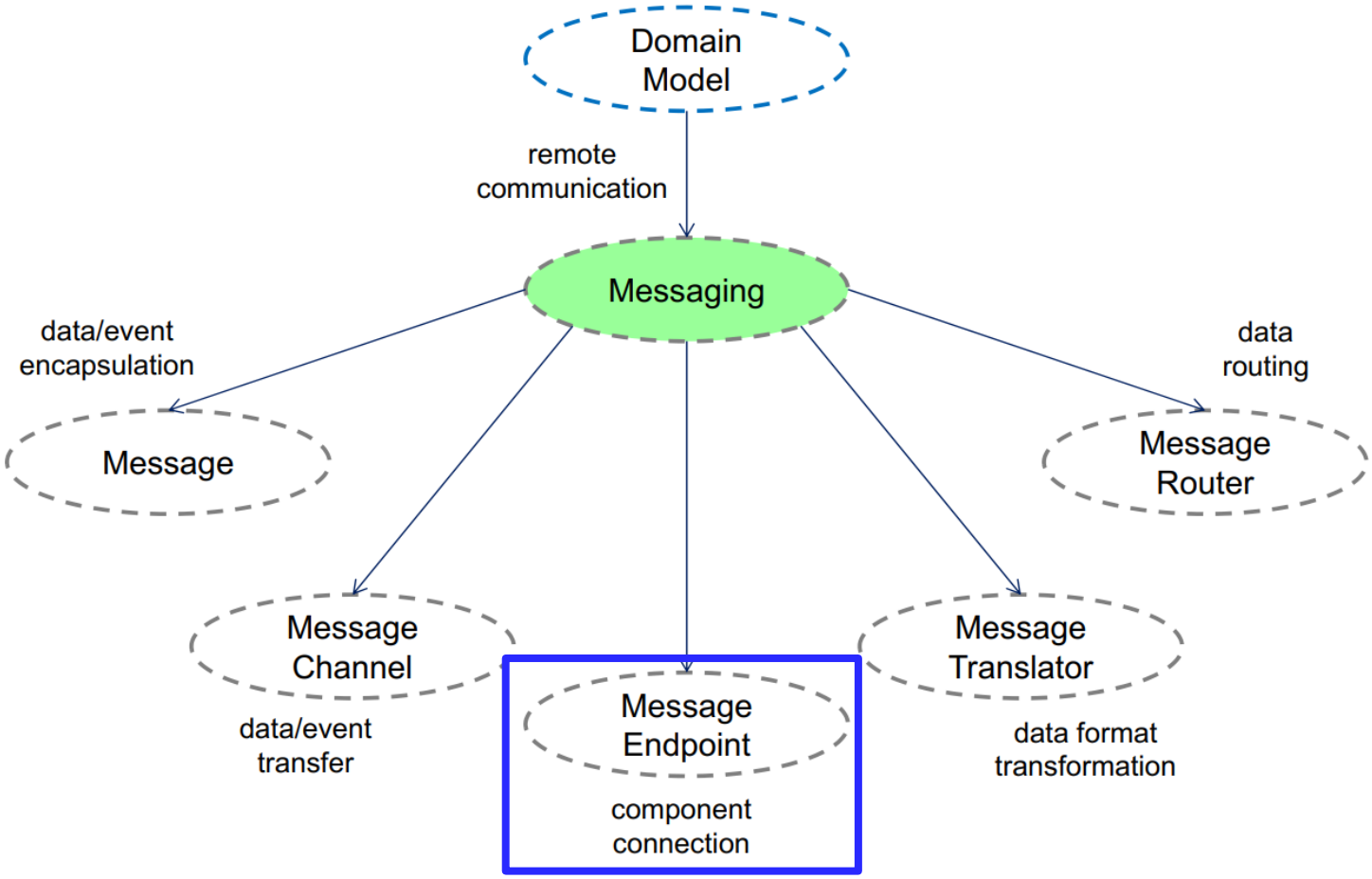
Message Channel

- Discussione
  - Ci sono diversi tipi comini di canali per messaggi
    - Un canale punto-punto (point-to-point channel) connette un mittente con un solo destinatario, ovvero, garantisce che ogni messaggio venga ricevuto ed elaborato da un singolo destinatario.
    - Invece un canale publish-subscribe (publish-subscribe channel) consente ai publisher di effettuare il broadcast di eventi e messaggi a molto subscriber interessati
  - Altri tipi di di canali rappresentano modo comuni di usare i canali
    - Ad es., un canale per messaggi non validi (invalid message channel) serve a disaccoppiare la gestione di messaggi errati dal resto della logica applicativa.

Message Channel

- Conseguenze
  - **sostiene un accoppiamento debole**
    - conoscere un canale per messaggi “intermediario” condiviso con un altro componente è una forma di accoppiamento più debole che non conoscere l’identità dell’altro componente
  - è possibile **assegnare** ad un canale la **responsabilità** per alcuni **attributi di qualità**
    - ad es., il livello di affidabilità per la consegna dei messaggi – best effort, persistente o transazionale
  - la gestione di un message channel **richiede memoria e risorse di rete**, nonché eventualmente anche **memoria persistente**

Ulteriori pattern di supporto



Message Endpoint

- Problema
  - si vogliono far comunicare, mediante lo scambio di messaggi, componenti o servizi autonomi (in particolare, anche preesistenti), ma non è possibile una loro comunicazione diretta – ad es., in questi componenti potrebbe non essere prevista la comunicazione basata sullo scambio di messaggi
  - per quanto possibile, si vuole sostenere un accoppiamento basso tra questi componenti e la soluzione tecnologica
    - certamente, non si vogliono accoppiare questi componenti agli elementi della soluzione di messaging – la tecnologia, i messaggi, i canali per messaggi, ...
  - se possibile, non si vogliono nemmeno accoppiare questi componenti tra loro – spesso, infatti, non si possono (oppure non si vogliono) modificare questi componenti
- è tuttavia necessario abilitare questi componenti o servizi all'invio e/o alla ricezione di messaggi

Message Endpoint

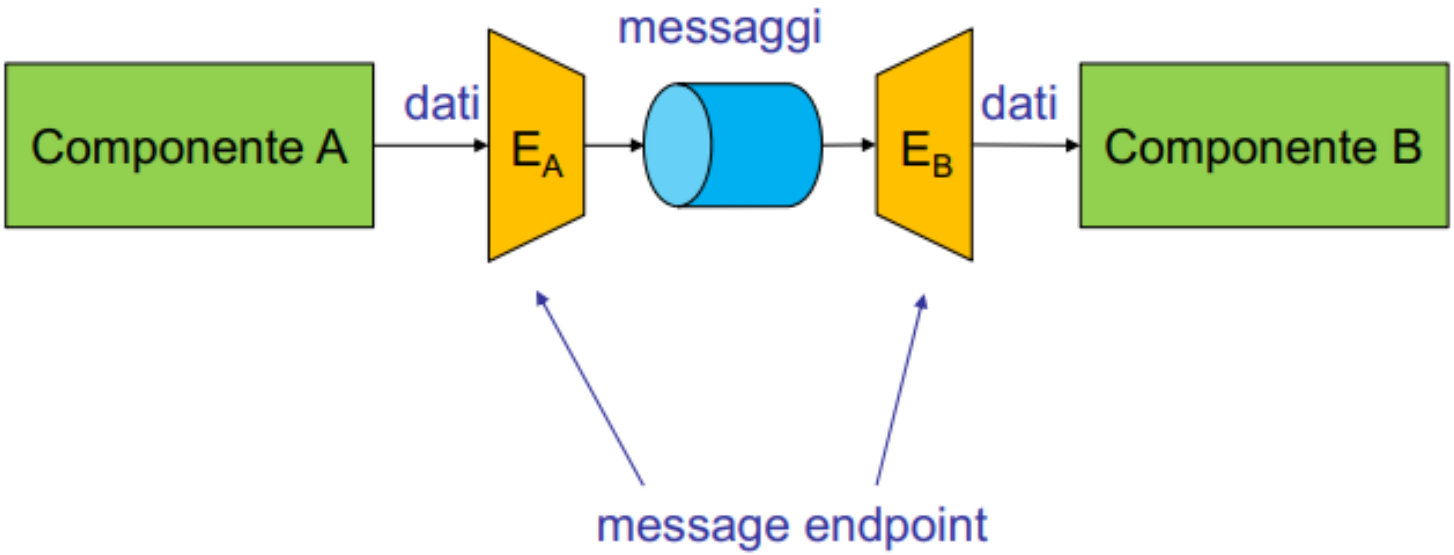
- Soluzione
  - Connetti i componenti e servizi che devono interagire all'infrastruttura di comunicazione mediante dei **message endpoint** («estremità» per messaggi) specializzati che gli consentono di scambiare messaggi

Message Endpoint

- Soluzione
  - quando un componente produttore deve comunicare dei dati, li passa al suo **message endpoint**
    - l'**endpoint** converte i dati in un messaggio, e poi inoltra il messaggio in un **message channel**
  - inoltre, il messaggio non viene ricevuto direttamente da un componente consumatore – piuttosto, viene ricevuto da un altro message endpoint
    - questo endpoint estrae i dati dal messaggio, e poi li passa, in un formato appropriato, al consumatore che li può utilizzare
  - talvolta, il componente produttore non comunica direttamente con il suo message endpoint – piuttosto, è l'endpoint che intercetta i dati del produttore

Message Endpoint

- Un esempio di comunicazione mediante endpoint



Message Endpoint

- **un message endpoint è un connettore che incapsula certamente tutto il codice per l'accesso alle API del message broker – in questo modo componenti e servizi non sono accoppiati alla specifica tecnologia del message broker utilizzato**
  - un message endpoint che ha solo questa finalità è anche chiamato un **messaging gateway**
  - usando un messaging gateway, i componenti possono essere a conoscenza dell'esistenza di un'infrastruttura per lo scambio di messaggi – ma il messaging gateway protegge questi componenti dai dettagli della tecnologia utilizzata, che rimangono dunque trasparenti ai componenti
- di solito un message endpoint ha lo scopo di catturare le interazioni già previste dai componenti e servizi di interesse e di ricondurle ad interazioni basate sullo scambio di messaggi

Message Endpoint

Un’altra tipologia comune di message endpoint è un **channel adapter**

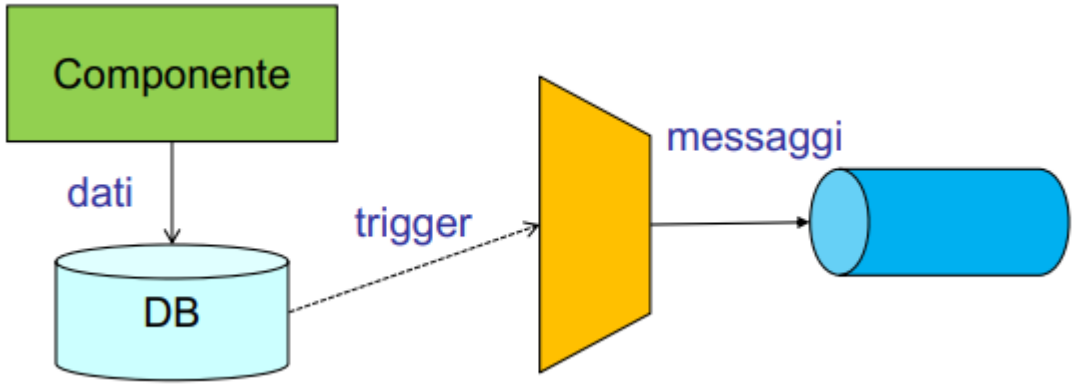
- un channel adapter ha lo scopo di nascondere completamente a un componente, servizio o applicazione l’infrastruttura per lo scambio di messaggi
- ovvero, l’introduzione di un channel adapter ha lo scopo di fare in modo che un componente o servizio non conosca nemmeno che viene utilizzato nell’ambito di una soluzione basata su messaggi

- molti elementi che fungono da “collante” in un sistema di integrazione di applicazioni, per “collegare” degli elementi preesistenti, sono proprio dei channel adapter

Esempio 1

In un'applicazione per basi di dati

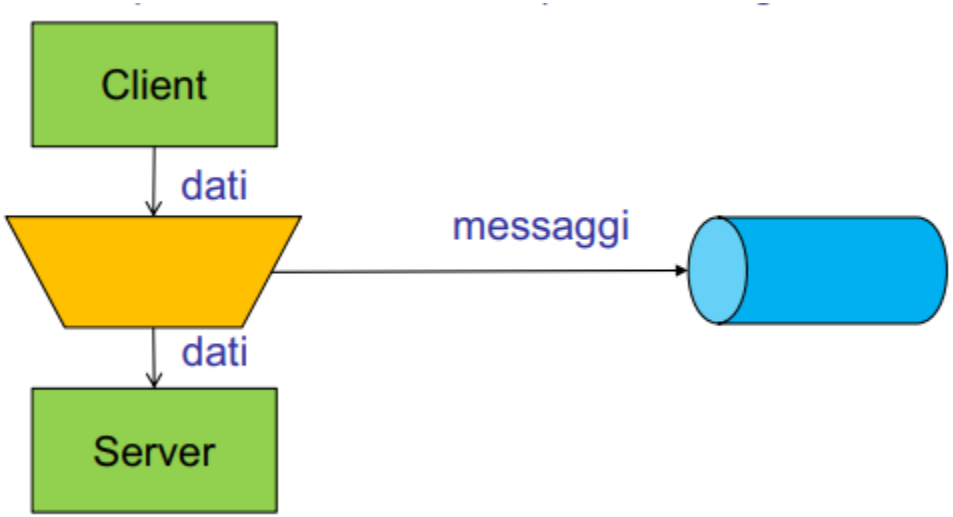
- un channel adapter potrebbe essere basato su un trigger per catturare un particolare cambiamento nella base di dati (ad es., "è stato memorizzato un nuovo ordine") – l'adapter si attiva per generare e trasmettere degli opportuni messaggi
- non è necessario modificare il componente che inserisce gli ordini nella base di dati



Esempio 2

In un'applicazione client-server

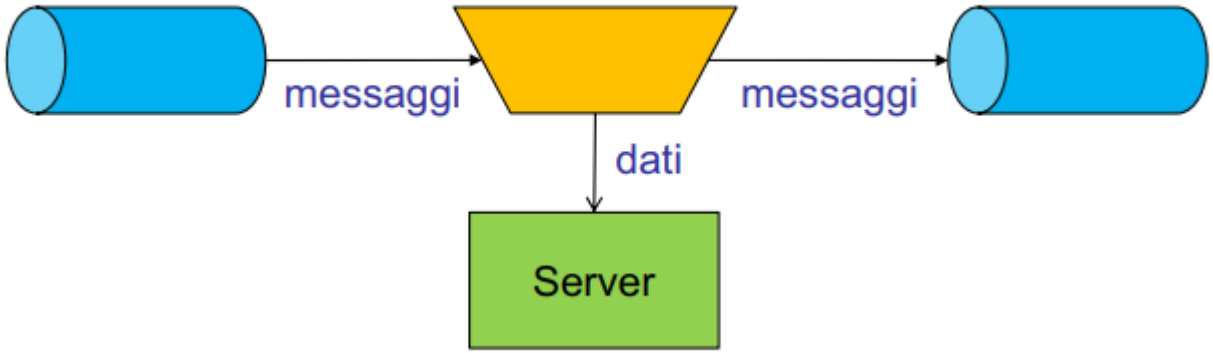
- un channel adapter si potrebbe interporre tra i client e un server – intercetta le richieste inviate al server, e le utilizza per generare ed inviare messaggi
- non è necessario modificare né i componenti client né il componente che implementa il servizio (ma il collegamento tra di essi sì)



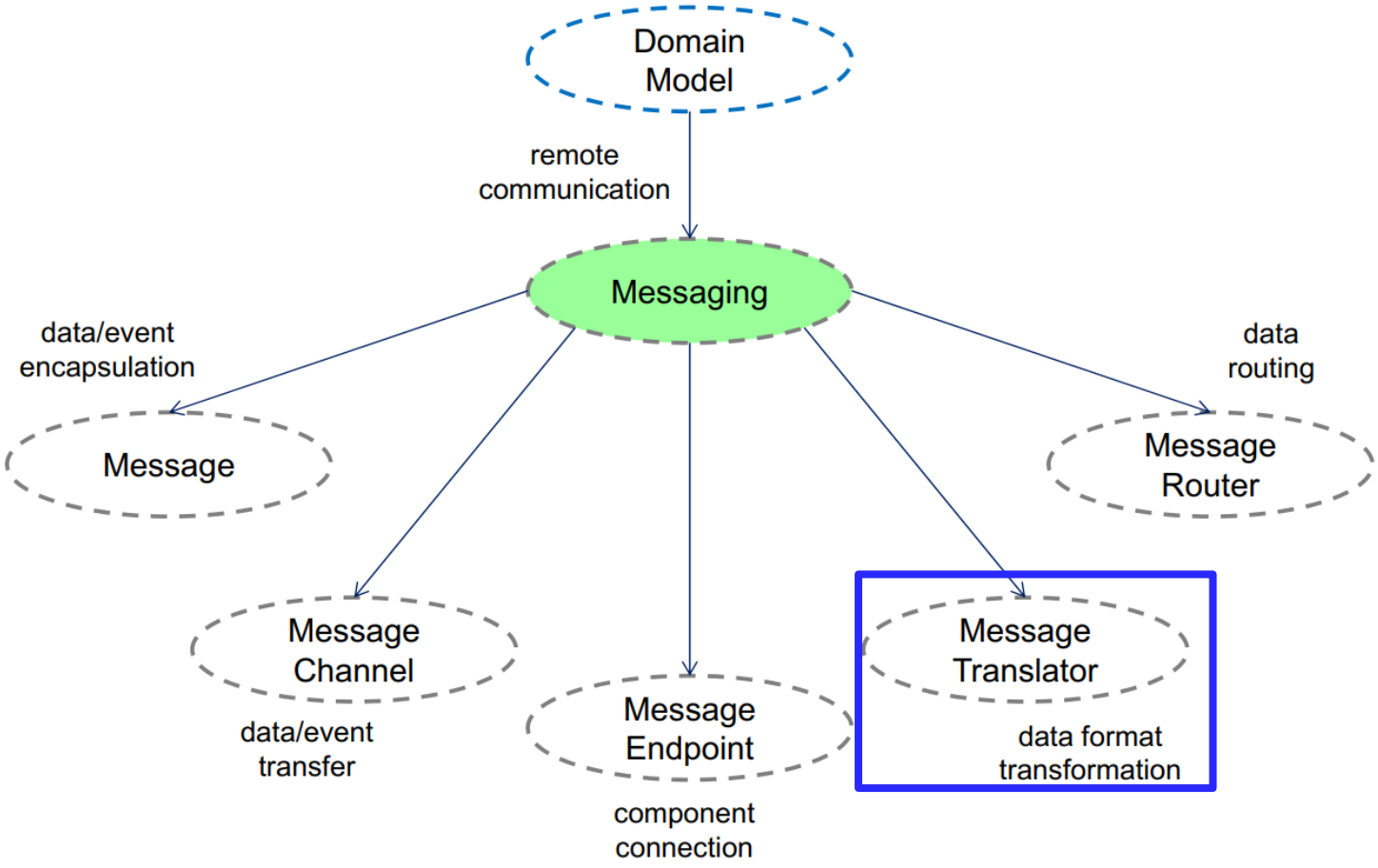
Esempio 3

In un'applicazione client-server

- un channel adapter potrebbe ricevere un messaggio relativo a un ordine, per verificare la disponibilità del prodotto ordinato – si comporta da client di un servizio di inventario, e comunica l'esito della verifica tramite un messaggio inviato a un altro componente
- non è necessario modificare il componente che implementa il servizio richiesto



Ulteriori pattern di supporto



Message Translator

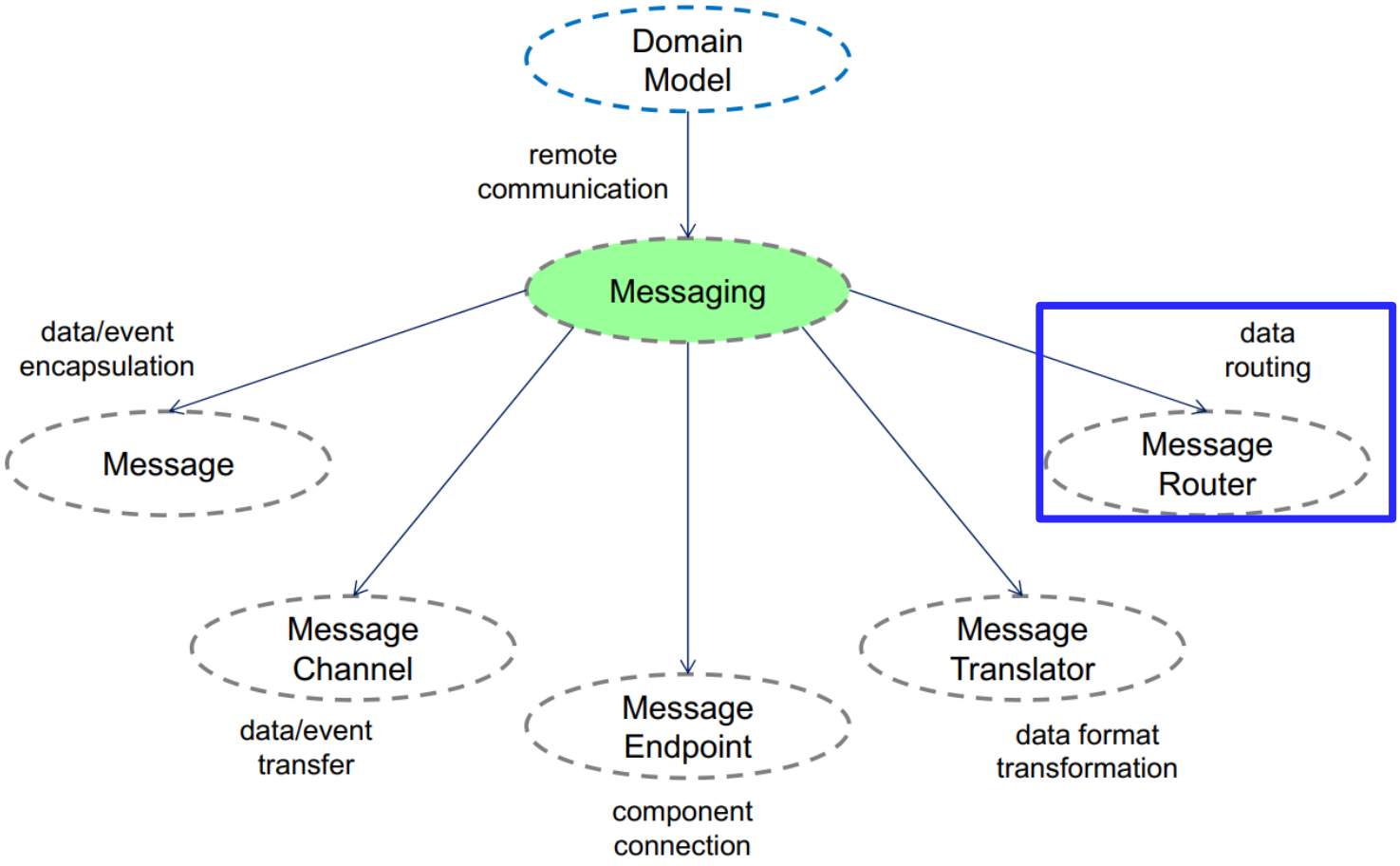
- Problema
  - per sostenere un accoppiamento debole, non è sempre possibile ipotizzare che i componenti consumatori che ricevono i messaggi comprendano il formato dei messaggi utilizzato dai componenti produttori che generano tali messaggi
  - in questi casi, è spesso necessario trasformare i messaggi dal formato utilizzato dai produttori iniziali al formato compreso dai consumatori finali

Message Translator

- Soluzione
  - introduci dei traduttori di messaggi (**message translator**) tra componenti o servizi, in grado di convertire messaggi da un formato all'altro
  - il produttore invia un messaggio nel formato che preferisce
  - il message translator garantisce che il consumatore riceva il messaggio nel formato a lui preferito
  - è anche possibile definire dei message translator che realizzano una traduzione bidirezionale tra formati di messaggi

  - Esempio: REST/SOAP translator

Ulteriori pattern di supporto



Message Router

Problema

- i messaggi scambiati tra componenti e servizi devono essere instradati nell'infrastruttura per i messaggi
- componenti e servizi (e anche canali e traduttori) non dovrebbero avere conoscenza del cammino di instradamento da utilizzare
- è tuttavia necessario scegliere un percorso per la propagazione dei messaggi

Soluzione

- introduci dei **message router** che consumano messaggi da un canale e li re-inseriscono in altri canali, sulla base di alcune condizioni – ad es., sull'header o sul contenuto del messaggio
- un message router connette un insieme di canali per messaggi in una rete di canali per messaggi – per muovere ciascun messaggio verso il consumatore/destinatario più opportuno

Integrazioni di applicazioni

- Le applicazioni interessanti vivono raramente in isolamento, piuttosto, devono spesso comunicare tra loro, per scambiarsi dati o servizi
  - In particolare, l'integrazione di applicazioni, ha l'obiettivo di comporre i componenti di un insieme di applicazioni esistenti per realizzare una nuova applicazione, con un maggior valore di business.

Stili di integrazione

- Nel corso del tempo sono stati introdotti e utilizzati in pratica diversi approcci per l'integrazione di applicazioni
  - **Trasferimento file**
    - Le applicazioni si scambiano dei file di dati condivisi
  - **Basi di dati condivise**
    - Le applicazioni memorizzano i dati che si devono scambiare in una base di dati condivisa
  - **Invocazioni remote**
    - Le applicazioni si scambiano i dati mediante l'invocazione di operazioni remote
  - **Messaging**
    - I sistemi si scambiano dati, sotto forma di messaggi e in modo asincrono, collegandosi ad un'infrastruttura per lo scambio di messaggi

Stili di integrazione

- Nel corso del tempo sono stati introdotti e utilizzati in pratica diversi approcci per l'integrazi
  - **Trasferimento fil**
    - Le applicaz ondivisi
  - **Basi di dati co**
    - Le applicaz devono scambiare in un
  - **Invocazioni remote**
    - Le applicazioni si scambiano i da mediante l'invocazione di operazioni remote
  - **Messaging**
    - I sistemi si scambiano dati, sotto forma di messaggi e in modo asincrono, collegandosi ad **un'infrastruttura per lo scambio di messaggi**

RabbitMQ

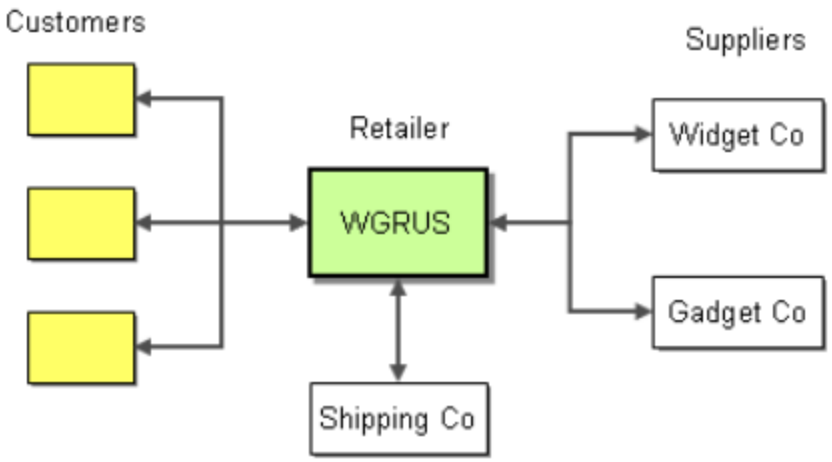
Stili di integrazione
limiti di ‘vecchie’ soluzioni

- In pratica, il Messaging è divenuto lo stile preferito per l’integrazione di applicazioni, poiché supera numerosi limiti degli altri approcci
  - **Trasferimento di file**
    - è poco tempestivo – spesso lo scambio di dati è tra singole coppie di applicazioni, con formati non compatibili con le altre applicazioni
  - **Basi di dati condivise**
    - È difficile progettare una base di dati condivisa per l’integrazione di molte applicazioni. Problemi di prestazione.
  - **Invocazioni remote**
    - Accoppiamento alto tra le applicazioni, l’integrazione sincrona è spesso caratterizzata da problemi di prestazioni e di affidabilità

Studio di caso
WGRUS

Studio di caso

- Viene presentato e discusso uno studio di caso per l'integrazione di applicazioni basata sul messaging
  - Widgets & Gadgets 'R Us è un rivenditore che acquista e rivende "widgets" e "gadgets"



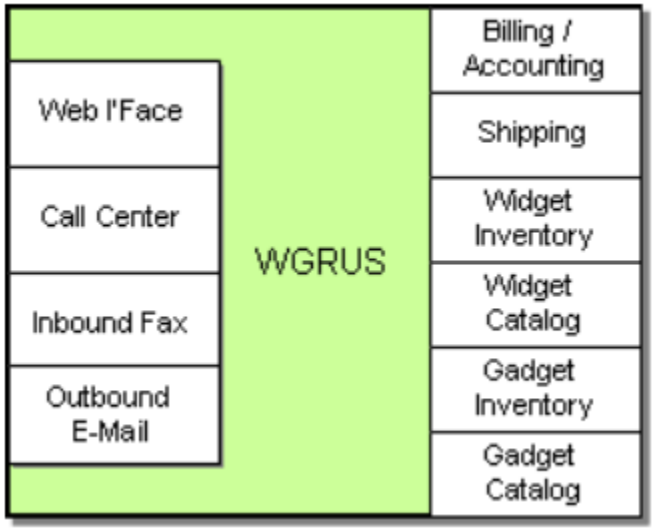
- **WGRUS** è il sistema software per Widgets & Gadgets 'R Us, che deve integrare alcuni componenti (preesistenti) dei sistemi informatici (preesistenti)

WGRUS

- Alcune funzionalità di WGRUS
  - Inserimento ordini
  - Elaborazione ordini
  - Verifica stato di avanzamento di un ordine
  - Gestione clienti
  - Gestione catalogo di prodotti
  - ...
- Qui consideriamo solo la gestione degli ordini
  - Inserimento, elaborazione, verifica stato di avanzamento degli ordini

WGRUS come problema di integrazione

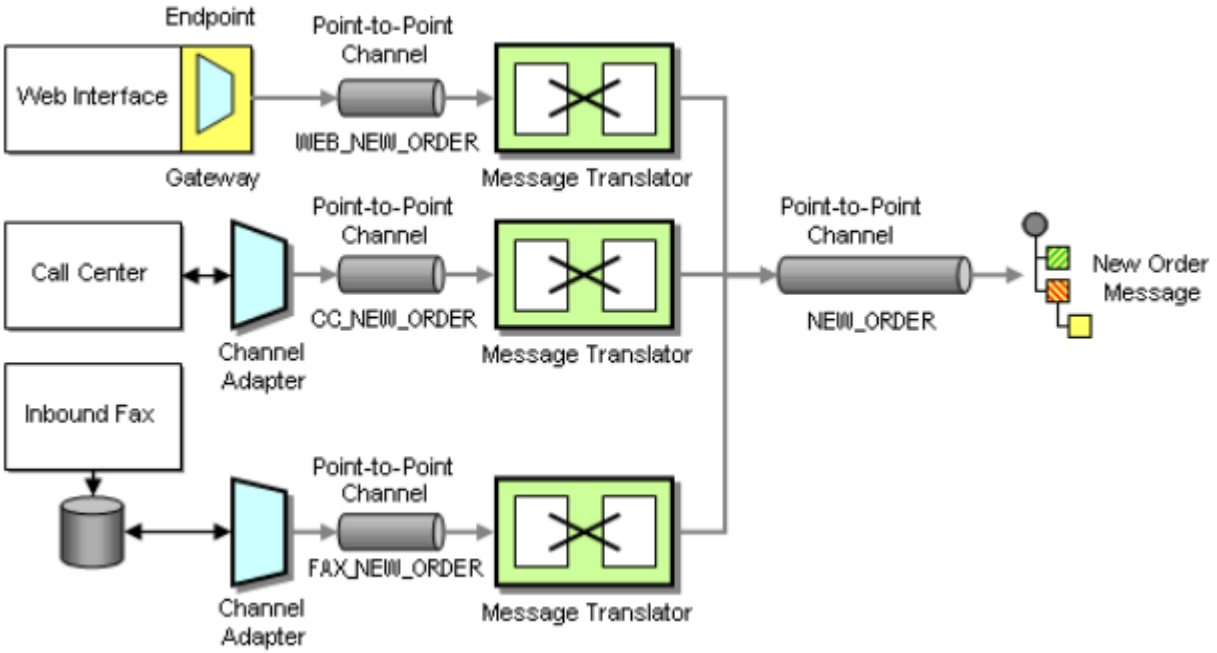
- Come detto, il sistema WGRUS deve realizzare le varie funzionalità integrando alcune componenti preesistenti
  - Tra cui i sistemi preesistenti a loro volta composti da vari elementi:



- A sinistra, sono mostrati i canali di interazione con i clienti
- A destra, i componenti applicativi preesistenti da riusare

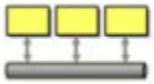
Ricezioni ordini

- Gli ordini possono essere ricevuti/immessi da vari client
  - Un client web, un client per un addetto al telefono, ordini ricevuti via fax, ciascuno genera ordini con un formato diverso
- Si vuole invece avere un flusso di messaggi (unico ed omogeneo) per tutti gli ordini



Pattern per l'integrazione (1)

- **Messaging**
  - Per integrare un insieme di applicazioni, in modo che possano scambiarsi informazioni e lavorare assieme



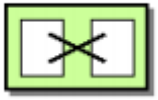
- **Message**
  - Un messaggio, ovvero, un tipo/flusso di messaggi

- **Message (point-to-point channel)**
  - Un canale (una coda) per lo scambio di messaggi

Pattern per l'integrazione (2)

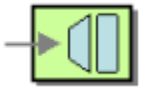
- **Message Endpoint**
  - Collega un componente al sistema di messaging, per trasmettere/ricevere messaggi

- **Message Translator**
  - Una trasformazione che cambia il formato di un messaggio

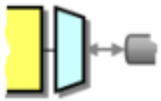


Pattern per l'integrazione (3)

- **Message Gateway**
  - Message endpoint che incapsula l'accesso al sistema di messaging, fornendo un'interfaccia con i metodi specifici del dominio applicativo, ma in modo indipendente dal sistema di messaging usato

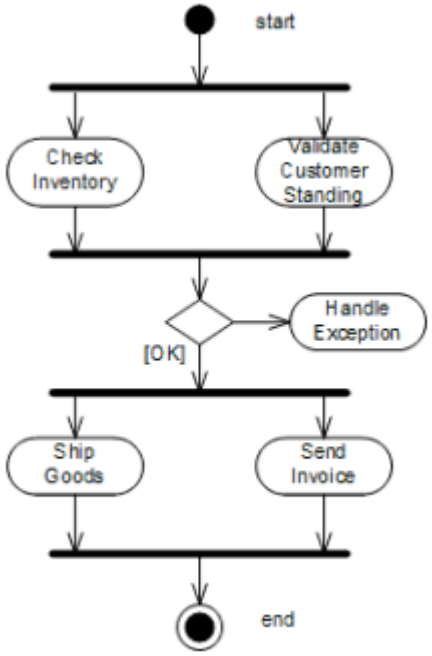


- **Channel Adapter**
  - Message endpoint che realizza una connessione tra un'applicazione (di solito preesistente) e il sistema di messaging



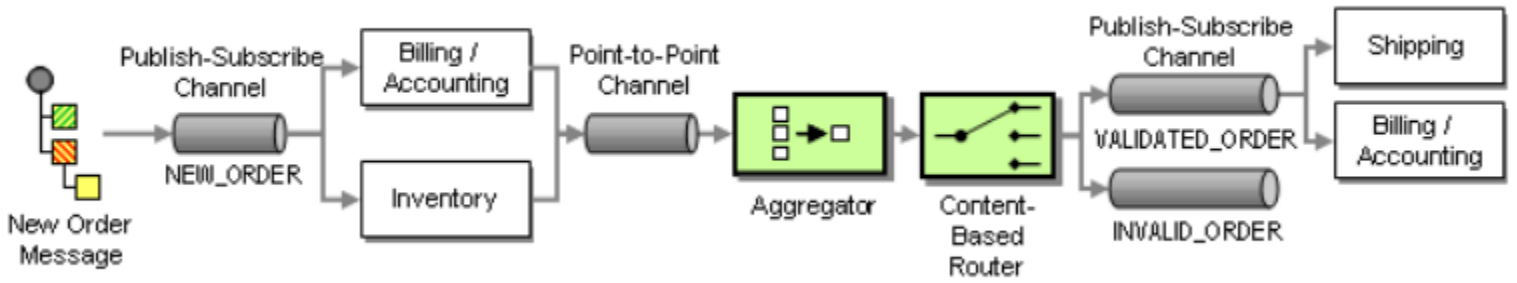
Elaborazione degli ordini

- Ora abbiamo un flusso consistente di ordini, l'elaborazione richiede
  - Verifica dello stato del cliente, nessun debito in sospeso
  - Verifica dell'inventario, disponibilità degli articoli ordinati
  - Se tutto ok, si può procedere



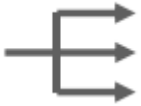
Elaborazione degli ordini

- Come elaborare gli ordini?
  - Ordini inviati separatamente e in parallelo a contabilità e inventario per le verifiche
  - Le due risposte devono poi essere aggregate
  - Gli ordini confermati vanno poi inviati ai sistemi di spedizione e fatturazione

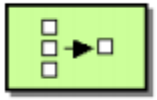


Pattern per l'integrazione (3)

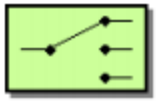
- **Publish-Subscribe Channel**
  - Un canale (un topic/argomento) per lo scambio di messaggi



- **Aggregator**
  - Combina il contenuto di messaggi diversi ma correlati

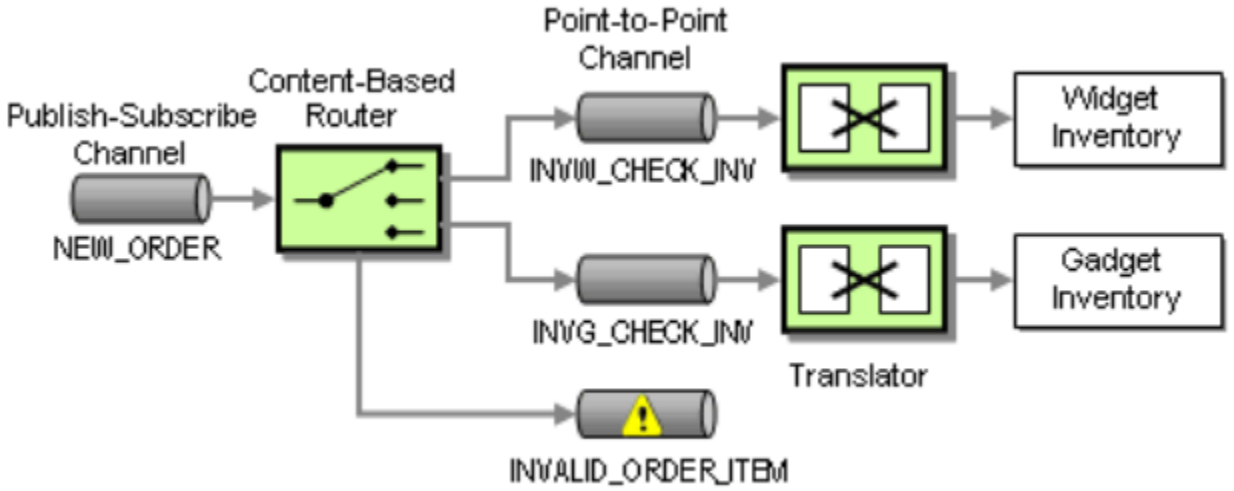


- **Content-based router**
  - Gira un messaggio a un'opportuna destinazione, sulla base del contenuto del messaggio



Controllo dell'inventario

- In realta, ci sono due sistemi/funzionalità per il controllo dell'inventario
  - Una per i i widget l'altra per i Gadget
  - Ciascuna richiesta va instradata al sistema giusto
    - Ipotesi (temporanea): una sola riga d'ordine per ordine
    - Ipotesi (semplificativa): il primo carattere del codice del prodotto è G o W

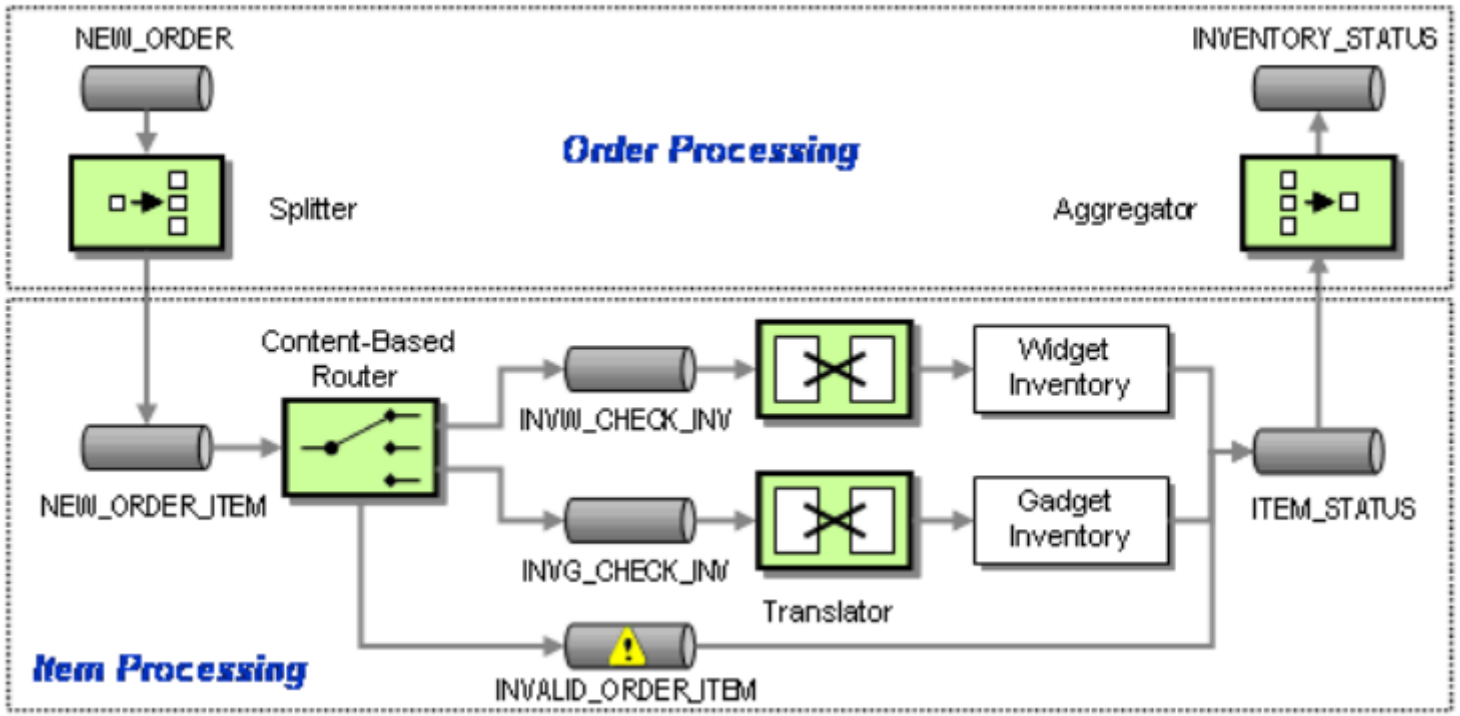


Pattern per l’integrazione (3)

- **Invalid Message Channel**
  - Destinazione di messaggi non validi

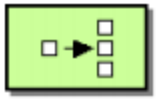
Ordini con più righe d'ordine

- In realta, un ordine contiene normalmente più righe d'ordine
  - Alcune saranno relative a widget, altre a gadget
  - La disponibilità delle merci va verificata riga d'ordine per riga d'ordine



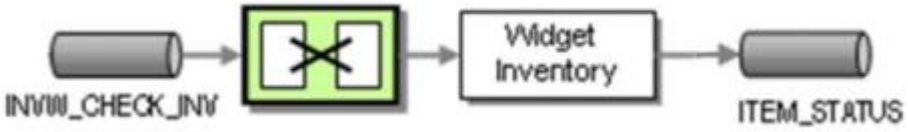
Pattern per l’integrazione (4)

- **Splitter**
  - Decompone un messaggio in un insieme di messaggi, ciascuno dei quali può richiedere (successivamente) una diversa elaborazione



Un’osservazione

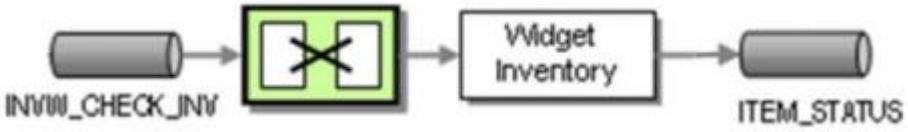
- Si consideri questa porzione del diagramma



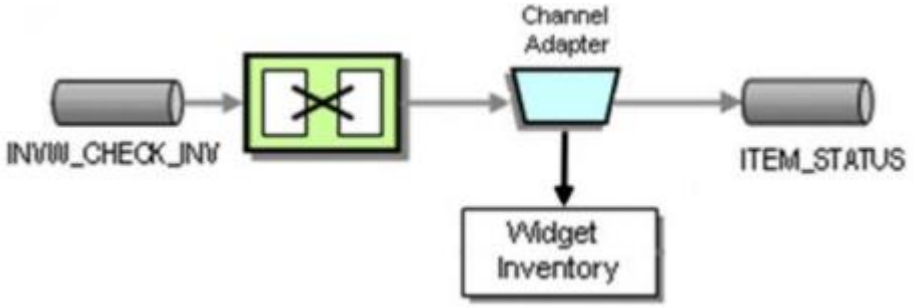
- Sembra che il componente preesistente Widget Inventory partecipi in prima persona alla soluzione di integrazione, me è proprio lui a consumare e produrre messaggi direttamente?

Un'osservazione

- Si consideri questa porzione del diagramma

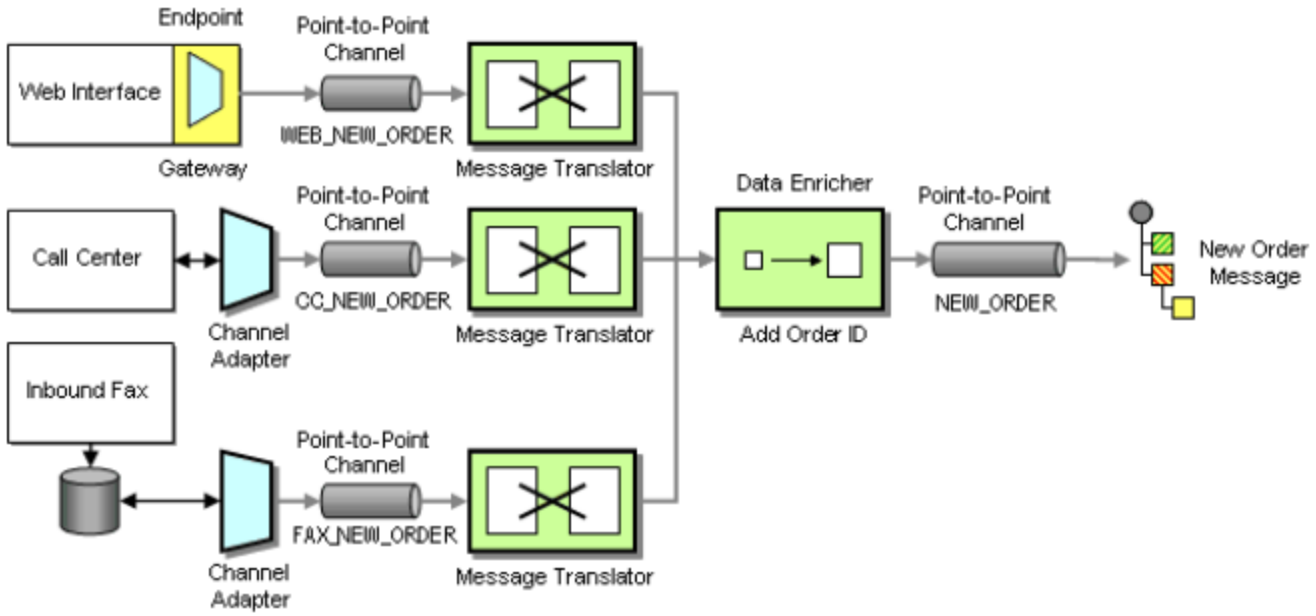


- Sembra che il componente preesistente Widget Inventory partecipi in prima persona alla soluzione di integrazione, me è proprio lui a consumare e produrre messaggi direttamente?
- No, tale componente verrà probabilmente utilizzato tramite un opportuno message endpoint (ad es., un channel adapter)



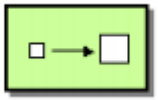
Identificatore d'ordine

- Messaggi elaborati separatamente possono essere ricombinati mediante un Aggregator sulla base di opportune informazioni di correlazione
  - Ad es, un identificatore d'ordine
  - Ma è necessario aggiungere un identificatore a ciascun ordine

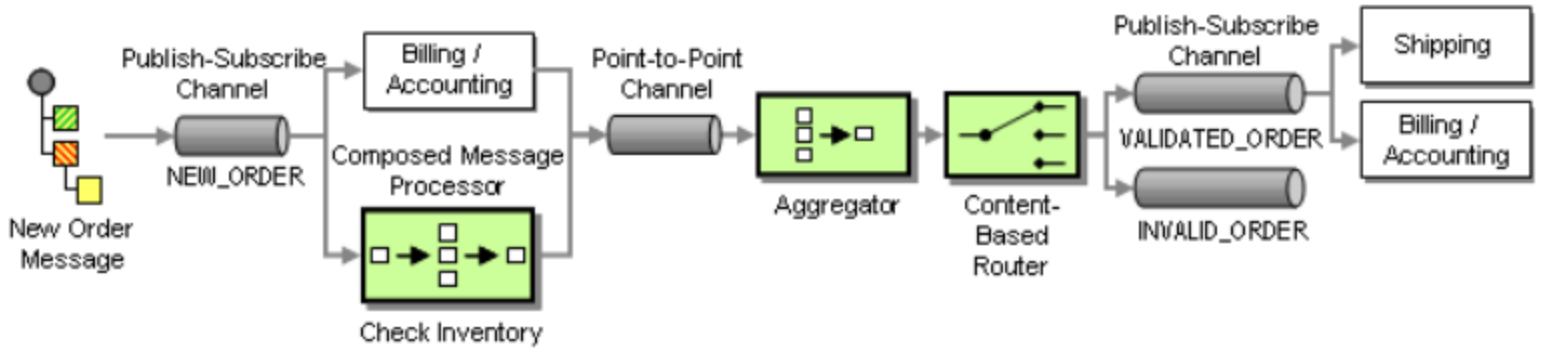


Pattern per l'integrazione (5)

- **Content Enricher**
  - Aggiunge informazioni a un messaggio

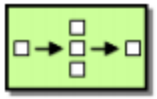


Gestione degli ordini



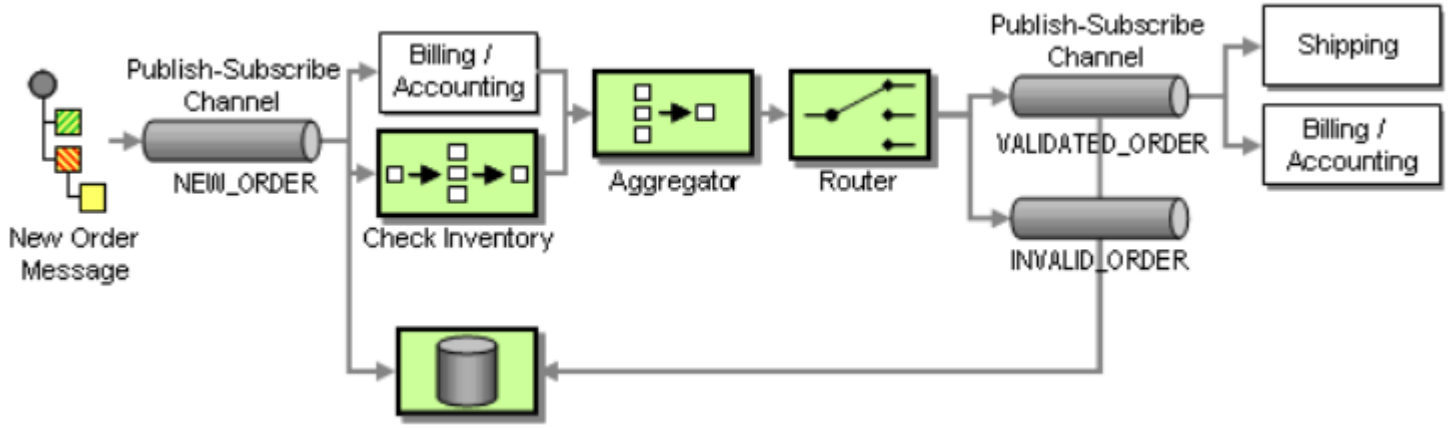
Pattern per l’integrazione (6)

- **Composed Message Processor**
  - Mantiene il flusso di messaggi complessivo, anche se i diversi messaggi richiedono elaborazioni diverse



Verificare lo stato di un ordine

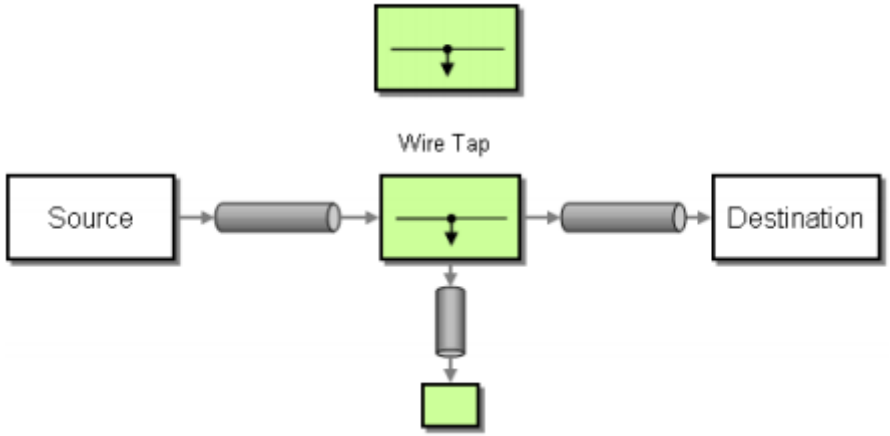
- L'elaborazione di un ordine richiede lo svolgimento di varie attività
  - Come è possibile verificare lo stato di avanzamento di un ordine? È stata effettuata spedizione? È in attesa di prodotti? È bloccato perché il cliente ha debiti in sospeso?
  - E' possibile rispondere conoscendo l'ultimo messaggio scambiato nel sistema circa l'ordine, questo può essere fatto memorizzando i messaggi rilevanti in un repository di messaggi



Pattern per l'integrazione (7)

- **Message Store**
  - Quando viene inviato un messaggio nel sistema, viene inviato anche un messaggio duplicato e memorizzato in un repository di messaggi
    - Semplice se il canale di cui bisogna memorizzare i messaggi è di tipo Publish-Subscribe

- **Wire Tap**
  - Per pubblicare su più canali i messaggi inviati su un certo canale



Discussione

- Lo studio di caso WGRUS esemplifica l'applicazione della comunicazione asincrona per l'integrazione di applicazioni, nonché l'applicazione di alcuni pattern di supporto allo scambio di messaggi
  - componenti preesistenti non vengono collegati tra di loro direttamente
  - piuttosto, l'integrazione è basata su costruzione, consumo, trasformazione, splitting, aggregazione e routing di messaggi, anche con riferimento a un certo numero di canali per messaggi
  - i componenti preesistenti non sono collegati direttamente nemmeno all'infrastruttura per lo scambio di messaggi – piuttosto, vengono collegati ad essa mediante dei message enpoint, che incapsulano l'accesso all'infrastruttura per i messaggi, ed inoltre fungono da "collante" tra i componenti preesistenti