UNIVERSITÀ DEGLI **STUDI**
DEL **SANNIO**
Benevento - A.A 2020/21

ADAPTIVE PRODUCER FRAMEWORK

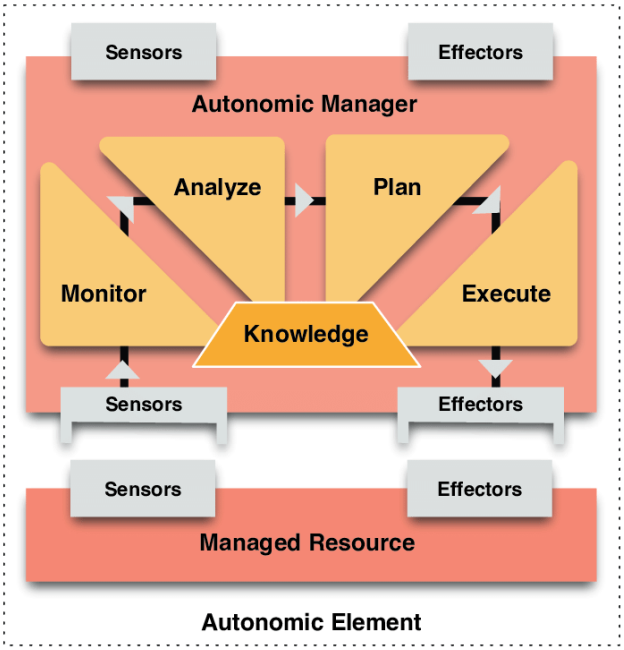
>> Corso di Laurea Magistrale in Ingegneria Informatica
>> Architettura e Sistemi Software Distribuiti
>> Progetto 6
>> Gruppo 1: Assunta De Caro, Pietro Vitagliano

Sommario

PREMESSE

Autonomic Computing

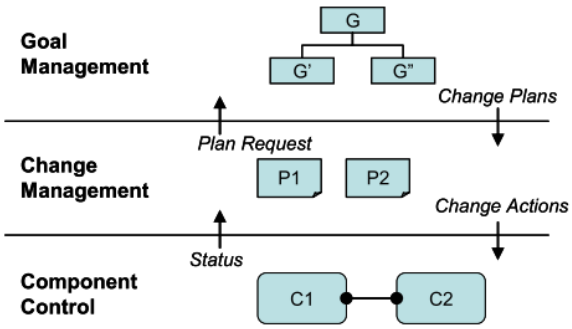
Sistemi software auto-adattivi, o autonomici, furono teorizzati per la prima volta da Paul Horn di IBM, ispirandosi al funzionamento dei neuroni umani. Il loro comportamento o la loro struttura deve adattarsi al mutare delle circostanze nel tempo, come la variabilità delle risorse disponibili o un aumento di richieste da gestire. "*Ciò implica, tra l'altro, monitorare il proprio comportamento rispetto ai propri obiettivi attuali (es. requisiti non funzionali) e modificare la propria struttura basata su una rappresentazione interna, ma esplicita, di sé.*" (Villegas, 2017). Questi sistemi, inoltre, devono esibire le proprietà di auto-configurazione, auto-correzione, auto-ottimizzazione e auto-protezione. Ne conviene che i vantaggi maggiori consistono nel ridurre i costi di dispiegamento, manutenzione e massimizzazione della disponibilità del sistema.



È necessario, quindi, comprendere a quali componenti architetturali assegnare le responsabilità di monitoraggio e interpretazione degli eventi, pianificazione della reazione e attuazione. Un manager autonomico implementa un loop di controllo per la gestione di risorse o altri elementi del sistema, ed è noto come **MAPE-K** (*Monitoring-Analysis-Planning-Execution and shared Knowledge*). Il Monitor raccoglie informazioni dai sensori e dall'osservazione dello stato interno dell'applicazione, corrispondente alle proprietà di QoS, e notifica solo i cambiamenti rilevanti. L'analizzatore decide se richiedere un adattamento del sistema, magari tenendo conto di più variazioni continue significative e generando un evento che le rappresenti. Il pianificatore, stimolato da questo trigger, riconfigura il sistema usando strategie ben note e materialmente applicate dall'esecutore. Infine, il gestore della conoscenza esplicita le informazioni pertinenti alla riconfigurazione dell'applicazione software.

Modelli e architetture

Una proposta di architettura autonomica avanzata da Kramer e Magee è il **modello di auto-gestione**, composto da tre layer, ciascuno dei quali definisce un insieme di responsabilità a diversi livelli di astrazione. Dal basso verso l'alto:

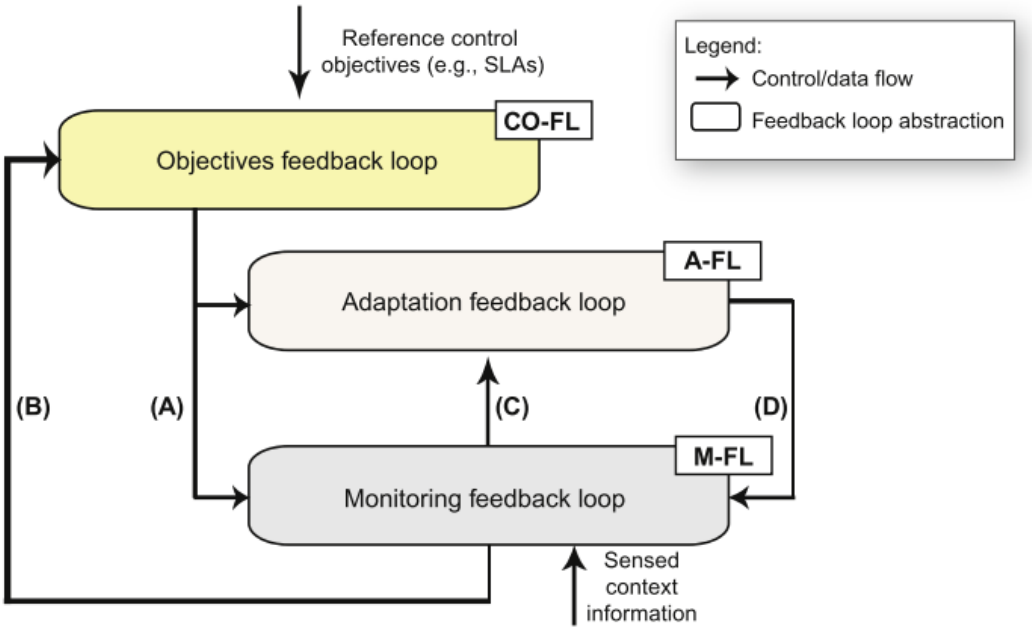


- Il **livello di controllo** implementa le funzionalità di feedback e attuazione della strategia di adattamento, sottoforma di algoritmi di autoregolazione. Può essere implementato secondo il modello MAPE-K.
- Il **livello di gestione del cambiamento** gestisce la configurazione decentralizzata, identifica le incongruenze nello stato del sistema e ristabilisce quello soddisfacente. Esegue piani di adattamento predeterminati. Ad esempio, in un sistema fault-tolerant, un

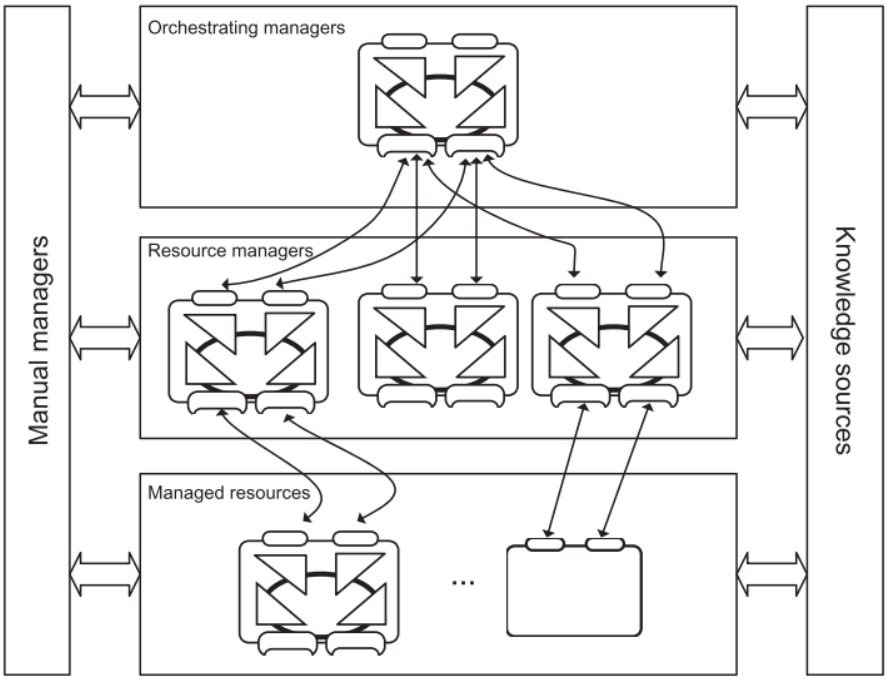
guasto potrebbe introdurre una replica attualmente passiva all'attivazione; questa azione è definita all'interno di questo livello.

- Il **livello di obiettivo** è responsabile del piano globale per raggiungere scopi di alto livello, tenendo in considerazione lo stato dei componenti correnti.

Il modello Dynamic Adaptive, Monitoring and Control Objectives (**DYNAMICO**) garantisce non solo l'adattamento comportamentale, ma del sistema di monitoraggio stesso, impostando ben tre cicli di controllo (sugli obiettivi, sull'adattamento e sul controllo).



L'architettura **ACRA** (Autonomic Computing Reference Architecture) è basata su una gerarchia a ciascuno dei quali controlla quello sottostante e può ospitare più componenti MAPE-K. A differenza del modello di auto-gestione definisce generiche responsabilità di orchestrazione e gestione delle risorse.



ANALISI DEL PROBLEMA

Obiettivi del Sistema

Realizzare un connettore (nodo sensore - piattaforma) che supporti una gestione dinamica della frequenza di raccolta dei dati. L'obiettivo è quello di evitare che si manifestino dei sovraccarichi dal punto di vista delle risorse. Allo scopo è possibile monitorare uno o più parametri prestazionali (es. dimensione delle code) e dimensionare dinamicamente il rate di trasmissione dei nodi sensore, utilizzando diverse politiche (es. aggregazione sul nodo, variazione di frequenza, drop di dati).

Vincoli e soluzioni tecnologiche

La coda da monitorare è gestita da un broker ActiveMQ Artemis che supporta i seguenti protocolli di comunicazione: JMS, MQTT, OpenWire, HornetQ, AMQP; la soluzione prevista dovrà offrire almeno l'implementazione basata su JMS e MQTT, tuttavia dovrà risultare estendibile nei confronti di altri protocolli.

JMS verrà impiegato anche per recuperare le informazioni fondamentali per monitorare il livello di congestione della destinazione e attivare le strategie di gestione. Per conoscere il numero di messaggi non ancora consumati è necessario contattare un indirizzo speciale di sistema, *activemq.management*, specificando il nome della destinazione da monitorare, l'operazione di gestione e i parametri. La documentazione di ArtemisMQ mette a disposizione tutti i dettagli per la connessione.

```
ClientSession session = ...
ClientRequestor requestor = new ClientRequestor(session, "activemq.management");
ClientMessage message = session.createMessage(false);
ManagementHelper.putAttribute(message, "queue.exampleQueue", "messageCount");
session.start();
ClientMessage reply = requestor.request(m);
int count = (Integer) ManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

Meccanismi di monitoraggio

Nel paragrafo precedente è stata introdotta la soluzione individuata al problema del monitoraggio, ma non è l'unica alternativa disponibile. Esistono diversi modi, oltre JMS, per accedere alla gestione di Apache ActiveMQ Artemis: usando JMX, Jolokia, oppure Prometheus.

JMX (Java Management Extension) è una tecnologia parte della Java Platform Standard Edition per la creazione di soluzioni distribuite, Web based, modulari e dinamiche per la gestione e il monitoraggio di dispositivi, applicazioni e reti basate sui servizi. A differenza di JMS prevede qualche accorgimento aggiuntivo, come abilitare la connector-port 1099 in *management.xml*, e settando correttamente la proprietà *-Djava.rmi.server.hostname* in *artemis.profile.cmd*. Si tenga sempre in considerazione che la versione di Artemis scelta è la 2.17.0. Per versioni precedenti di Artemis potrebbe richiedere ulteriori operazioni, disponibili sulla documentazione.

```
16 public class JMXProbe {
17     public static void main(String[] args) {
18         String jmx_url = "service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi";
19 
20         try {
21             ObjectName on = ObjectNameBuilder.create("org.apache.activemq.artemis", "0.0.0.0")
22                     .getQueueObjectName(SimpleString.toSimpleString("testQueue"),
23                             SimpleString.toSimpleString("testQueue"), RoutingType.MULTICAST);
24             HashMap<String, String[]> env = new HashMap<String, String[]>();
25             String[] creds = {"brokerProject", "brokerProject"};
26             env.put(JMXConnector.CREDENTIALS, creds);
27             JMXConnector connector = JMXConnectorFactory.connect(new JMXServiceURL(jmx_url), env);
28             MBeanServerConnection mbsc = connector.getMBeanServerConnection();
29             QueueControl queueControl = MBeanServerInvocationHandler.newProxyInstance(mbsc, on, QueueControl.class, false);
30             long messageCount = queueControl.getMessageCount();
31 
32             System.out.println("Queue contains " + messageCount + " messages.");
33 
34         } catch (Exception e) {
35             e.printStackTrace();
36         }
37     }
38 }
```

Jolokia utilizza un protocollo JSON-over-HTTP. La comunicazione si basa su un paradigma richiesta-risposta. Le richieste possono essere formulate in due modi: come HTTP GET, codificando parametri nell'URL, o come POST, inserendoli nel payload JSON. Viene ritornato un JSON dal formato invariato. Per esempio, contattando tramite browser http://username:password@localhost:8161/console/jolokia/read/org.apache.activemq.artemis:broker="0.0.0.0"/Version, si ottiene la seguente risposta:

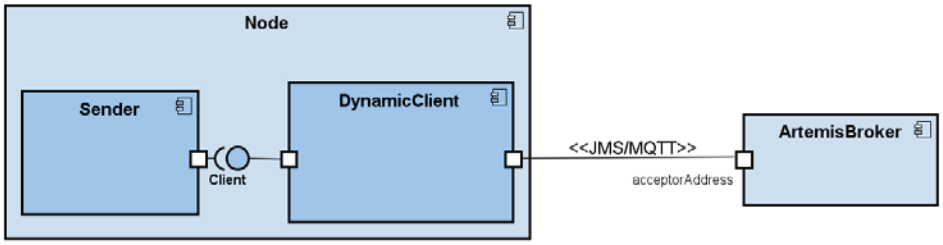
{"request":{"mbean":"org.apache.activemq.artemis:broker=\"0.0.0.0\"","attribute":"Version","type":"read"},"value":"2.0.0-SNAPSHOT","timestamp":1487017918,"status":200}

Prometheus è un software per il monitoraggio di sistemi scalabili di grandi dimensioni e la creazione di storici, il quale mette a disposizione un plug-in specifico già presente all'installazione dell'Artemis Broker. Per abilitarlo e iniziare a raccogliere le informazioni di interesse, come il parametro *message.count* bisogna seguire la procedura indicata sul sito di RedHat.

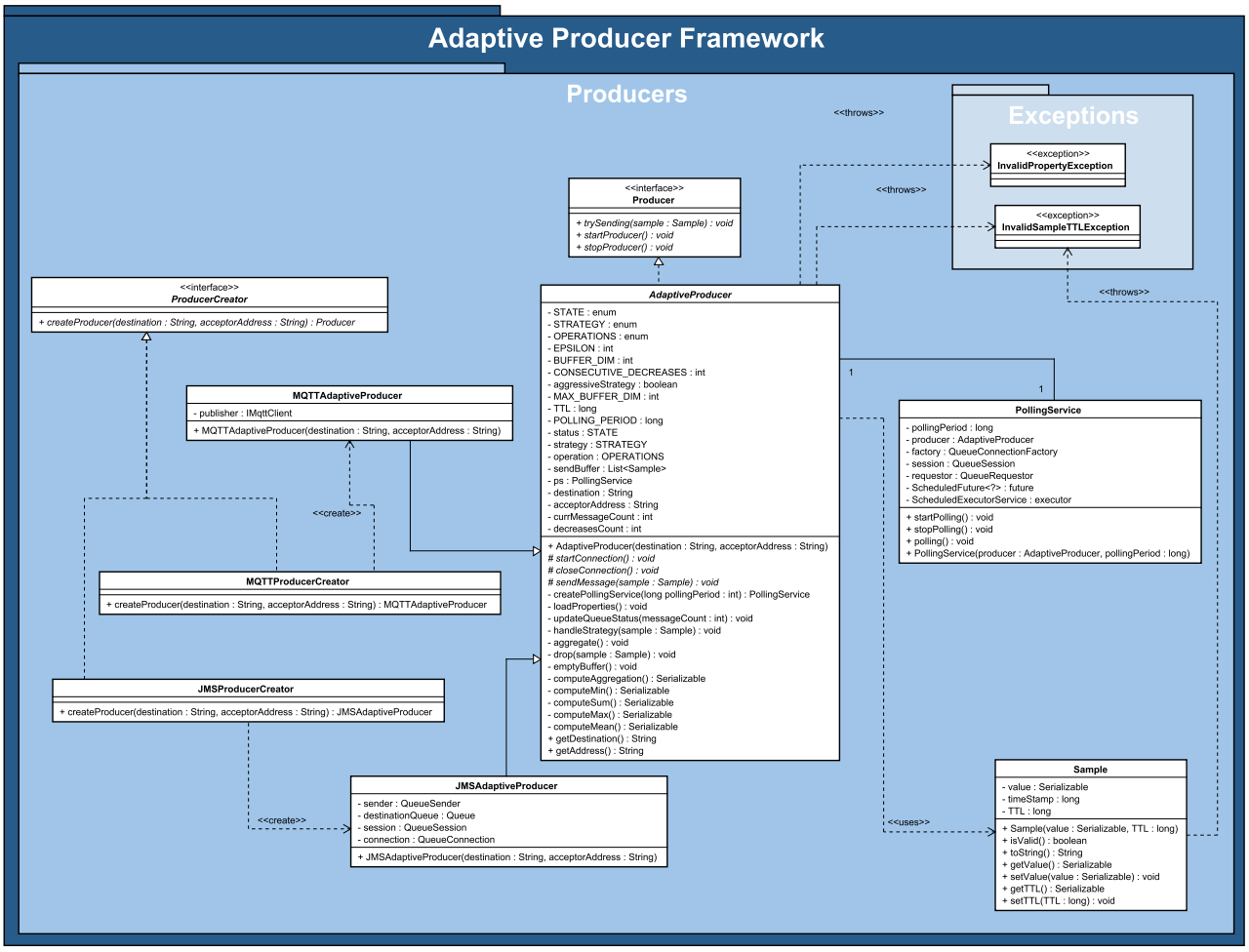
ARCHITETTURA DEL SISTEMA

La soluzione proposta corrisponde alla progettazione di un **framework OO black box**, configurabile dall'utente tramite un file di properties esterno al fine garantire il tuning dei valori di default ove fosse necessario.

L' Adaptive Producer dovrà incapsulare sia i meccanismi di comunicazione, sia quelli di gestione del rate di trasmissione, rendendoli totalmente trasparenti al nodo sensore e offrendogli un'interfaccia minimale per l'invio dei messaggi.



DESIGN CLASS DIAGRAM



Producer	É l'interfaccia implementata da Adaptive Producer. Prevede i metodi di avvio/terminazione del producer e invio dei messaggi.
Adaptive Producer	Classe astratta che fattorizza la gestione delle strategie e l'aggiornamento dello stato della coda, rendendoli uniformi per tutte le sue sottoclassi concrete. Implementa l'interfaccia Producer. Delega la scrittura dei metodi *sendMessage*, *startConnection*, *stopConnection* alle sue sottoclassi perché dipendenti dal protocollo di comunicazione utilizzato.
JMS Adaptive Producer	Classe concreta che estende Adaptive Producer, incapsulando gli oggetti necessari a gestire uno scambio dati basato su protocollo JMS.
MQTT Adaptive Producer	Classe concreta che estende Adaptive Producer, incapsulando gli oggetti necessari a gestire uno scambio dati basato su protocollo MQTT.
Producer Creator	Per nascondere la creazione di uno specifico Producer e ridurre l'impatto di un cambiamento sul nodo utilizzatore viene impiegato il pattern Creator. Questa è l'interfaccia della factory che delinea il metodo *create*. Le uniche informazioni necessarie alla creazione saranno il nome della coda/topic e l'indirizzo dell'accettore del broker.
JMS Producer Creator	Factory concreta di un JMS Adaptive Producer.
MQTT Producer Creator	Factory concreta di un MQTT Adaptive Producer.

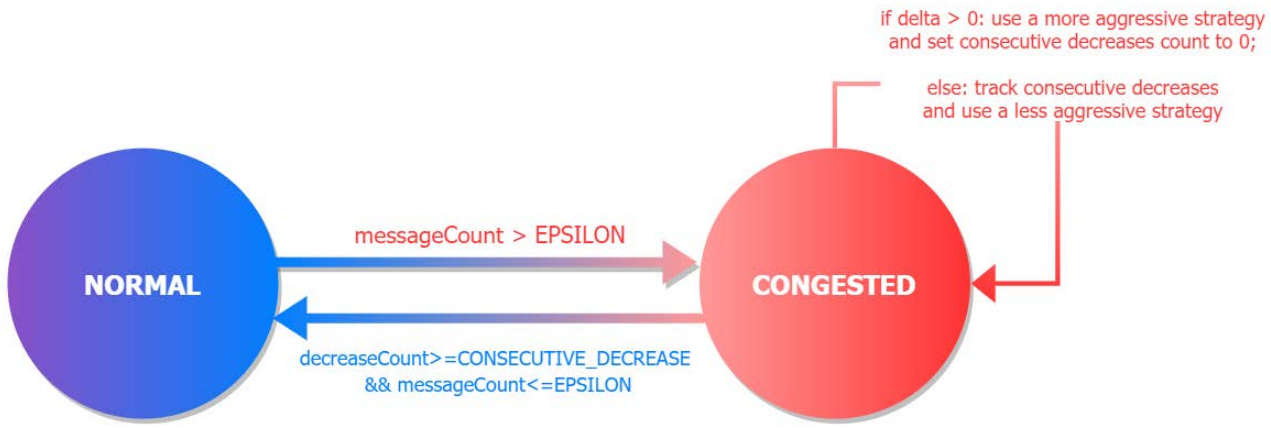
Polling Service	Recupera le informazioni sullo stato della coda e aggiorna periodicamente l'Adaptive Producer. L'interazione è basata su JMS.
Sample	Un oggetto di tipo Producer accetta l'inoltro di un Sample, che incapsula il payload del messaggio e richiede un Time To Live, scaduto il quale verrà considerato non valido per l'invio. Implementa l'interfaccia Serializable.

Strategie di gestione della congestione

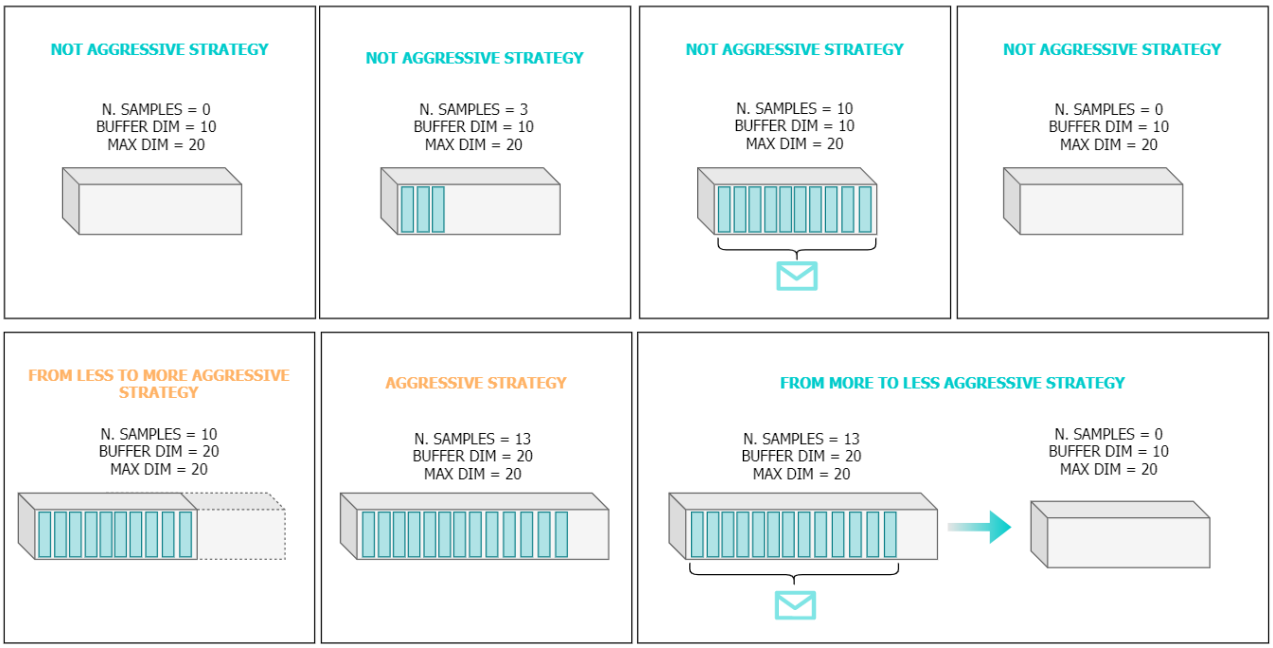
Con riferimento alla classe Adaptive Producer, andiamo a descrivere le scelte implementative adottate per la sua realizzazione.

Lo stato della coda può essere normale o congestionato. Si permane nel primo fin quando l'incremento dei messaggi nella coda non supera una soglia fissata. Per calcolare l'incremento basta confrontare il numero dei messaggi attuali con quello restituito al check precedente.

Quando ci si trova nello stato di congestione, si applica la strategia selezionata (drop o aggregable) e si monitora la loro variazione. In entrambi i casi, si ricorre a un buffer in cui memorizzare i sample e differire il loro invio, abbassando il rate del sender. Se la variazione è positiva, poiché la coda si sta riempiendo, si aumenta l'aggressività della strategia, ampliando il buffer. Quando si verificano un certo numero di decrementi consecutivi, si riduce l'aggressività della strategia, e, se il numero di messaggi nella coda è minore della soglia indicata, si ritorna nello stato normale.



Il comportamento del buffer merita una spiegazione dettagliata. All'avvio il buffer è vuoto, la sua dimensione minima è fissata dall'utente nel file di configurazione. Con l'aggressività della strategia manipoliamo il numero massimo di sample ammissibili al suo interno. Aumentandola raddoppiamo la grandezza del buffer, al contrario, ridurre l'aggressività significa dimezzarla. Quando il buffer si riempie bisogna svuotarlo: l'aggregazione prevede l'invio di un messaggio che corrisponde al valore calcolato applicando ai sample validi un'operazione tra quelle possibili (di media, massimo, minimo o somma). Con il drop ci limitiamo a inviare i campioni validi, essendo rivolto a dati non aggregabili. Nella figura sottostante, due possibili evoluzioni del buffer: una rappresentativa del ciclo di riempimento/svuotamento in condizioni invariate, l'altra descrivente l'estensione e la contrazione del buffer al cambiamento dell'aggressività.



Nello stato normale si procede direttamente con l'invio del sample corrente e di quelli eventualmente rimasti all'interno del buffer. A prescindere dalla situazione, se il buffer è riempito per più della metà, verrà inviata solo una parte dei sample validi, per evitare una scarica eccessiva che potrebbe generare o peggiorare la congestione. I rimanenti verranno inoltrati all'invio del prossimo sample. Un sample è ritenuto valido finché non è scaduto il suo time to live.

I metodi attivamente coinvolti nella gestione della congestione e la cui logica è definita in Adaptive Producer sono i seguenti:

TrySending	Prova ad inviare il sample accettato come parametro controllando lo stato della congestione. Se è NORMAL, procede con l'invio ed eventuale svuotamento del buffer, altrimenti richiama il metodo *handleStrategy* per applicare la strategia richiesta dall'utente.
HandleStrategy	A seconda della strategia richiama i metodi *drop* o *aggregable*.

Drop	Se il buffer non è pieno, aggiunge il sample per differirne l'invio, altrimenti provvede prima a svuotarlo.
Aggregate	Se il buffer non è pieno, provvede a memorizzare il sample in attesa che si riempia, si possa effettuare l'aggregazione sui soli campioni validi e, al temine, si elimini il suo contenuto. Richiama *computeAggregation*.
Compute Aggregation	Ritorna il valore ottenuto dall'aggregazione a seconda dell'operazione indicata dall'utente (minimo, massimo, somma, media). Se non ci sono valori validi ritorna un oggetto nullo.
Update QueueStatus	Aggiorna lo stato della coda in base al valore dei messaggi contenuti in essa e alla variazione rispetto al controllo precedente. Infine, aggiorna la capacità del buffer a seconda del livello di aggressività, modellando un diverso comportamento in base alla strategia.
EmptyBuffer	Svuota il buffer; se il numero di campioni è maggiore alla metà della capacità, ne invia solo una parte, altrimenti tutti. Ovviamente, i sample inviati devono essere validi.

Gestione della connessione e polling

L'Adaptive Producer deve concretizzare i metodi richiesti dall'interfaccia Producer relativi all'avvio e alla terminazione del servizio. L'operazione di *startProducer* si occupa di avviare il servizio di polling, invocando il metodo *startPolling* sull'istanza di PollingService, ed eseguire eventuali operazioni preliminari per la comunicazione, le quali differiscono a seconda del protocollo e sono delegate alle sue sottoclassi. Pertanto, i metodi *startConnection* e *stopConnection* saranno dichiarati astratti e protected. JMSAdaptiveProducer, ad esempio, provvederà a creare ed avviare una connessione, una sessione e un sender. Analogamente, MQTTAdaptiveProducer inizializzerà un publisher e una connessione.

Il polling verrà eseguito da un thread a cadenza regolare, richiamando sull'istanza del Adaptive Producer il metodo di aggiornamento dello stato, *updateQueueStatus*, dopo aver recuperato le informazioni sulla destinazione. Ricordiamo che PollingService sfrutta un connettore JMS per contattare il broker.

Dualmente a quanto descritto, il metodo *stopProducer* svuota definitivamente il buffer, richiama *stopPolling*, ed esegue le operazioni necessarie a terminare la comunicazione.

Configurazione del sistema

All'interno del folder Resources è possibile accedere al file "config.properties" che contiene le proprietà modificabili dall'utente per adattarle a contesti specifici.

PROPRIETÀ	**DESCRIZIONE**	**VALORE DI DEFAULT**
epsilon	Soglia di default, superata la quale si segnala una congestione.	5
consecutive Decreases	Numero di decrementi consecutivi necessari a ridurre l'aggressività della strategia; abbinato ad un numero di messaggi in coda inferiore ad epsilon, permette di tornare nello stato normale.	5
bufferDim	Grandezza minima di partenza del buffer.	10
pollingPeriod	Periodo di polling per aggiornare lo stato della coda. Espresso in ms.	1000
strategy	Strategia scelta per gestire la congestione: drop o aggregable.	AGGREGABLE
aggregationType	Operazione di aggregazione nel caso sia selezionata l'omonima strategia. È possibile optare per: mean, min, max, sum.	MEAN
TTL	Time to live da assegnare ai messaggi che contengono il valore computato tramite l'aggregazione. Espresso in ms.	1000

PRESTAZIONI

Per verificare il funzionamento del framework è stata implementata una versione di test, sia JMS che MQTT, che consente di simulare un'interazione e di monitorarla dal punto di vista di un nodo. La frequenza di consumo e trasmissione è generata casualmente entro un range da 500 a 3000 ms. È stata introdotta una classe Metric per tenere traccia del numero di congestioni in una simulazione, il periodo di congestione più duraturo, il numero medio di messaggi in coda, il tempo medio di congestione totale. Per ciascuna strategia sono state effettuate 10 simulazioni da 1 min con producer JMS, prima con un rapporto tra sender/consumer di 1:1, 2:1 e 3:1. Il file di configurazione mantiene i valori di default.

CT=Congestion Times
TCT = Total Congestion Time
TNT = Total Normal Time
LCP = Longest Congestion Time
MQM = Mean Queued Messages
TCT/ST = Total Congestion Time / Simulation Time

DROP 5 senders / 5 consumers

N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	6	27,11	32,89	8,12	3,80	45,18
2	0	0,00	60,00	0,00	6,93	0,00
3	2	6,14	53,86	3,09	2,40	10,23
4	5	22,38	37,62	10,21	3,38	37,31
5	4	10,65	49,35	3,05	3,40	17,76
6	8	27,34	32,66	5,08	2,83	45,57
7	1	3,04	56,96	3,04	0,67	5,07
8	4	12,17	47,83	3,06	2,22	20,29
9	0	0,00	60,00	0,00	1,30	0,00
10	5	15,19	44,81	3,05	2,62	25,32
MEAN	**3,50**	**12,40**	**47,60**	**3,87**	**2,95**	**20,67**

DROP 10 senders / 5 consumers

N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	5	30,37	29,63	16,20	4,60	50,61
2	6	42,53	17,47	27,35	6,55	70,89
3	6	27,34	32,66	12,16	3,48	45,57
4	5	18,25	41,75	6,08	3,62	30,42
5	6	38,43	21,57	22,28	5,35	64,05
6	4	20,27	39,73	10,15	3,20	33,78
7	5	27,95	32,05	16,25	4,63	46,58
8	4	25,29	34,71	15,18	3,47	42,15
9	5	21,26	38,74	9,12	3,07	35,43
10	4	33,46	26,55	21,30	4,82	55,76
MEAN	**5,00**	**28,51**	**31,49**	**15,61**	**4,28**	**47,52**

AGGREGABLE 5 senders / 5 consumers

N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	1	3,04	56,96	3,04	2,87	5,07
2	4	12,15	47,85	3,05	3,07	20,25
3	6	18,20	41,80	3,05	3,55	30,33
4	6	15,76	44,24	3,05	3,00	26,26
5	4	14,21	45,79	5,08	2,33	23,68
6	6	22,40	37,60	5,07	3,80	37,33
7	4	14,19	45,82	5,09	3,13	23,64
8	7	19,56	40,444	3,05	2,92	32,59
9	6	23,31	36,69	8,15	3,1	38,84
10	7	19,88	40,12	3,05	2,68	33,13
MEAN	**5,1**	**16,27**	**43,73**	**4,17**	**3,04**	**27,11**

AGGREGABLE 10 senders / 5 consumers

N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	1	32,45	27,55	32,45	5,67	54,09
2	3	16,18	43,82	10,11	3,90	26,97
3	4	23,29	36,71	14,15	4,00	38,81
4	1	16,20	43,80	16,20	3,73	27,00
5	4	29,37	30,63	19,23	2,95	48,95
6	3	18,22	41,78	12,15	3,77	30,37
7	6	22,25	37,75	5,07	2,87	37,09
8	3	19,24	40,76	11,17	4,10	32,06
9	5	28,33	31,67	16,17	4,20	47,22
10	1	11,14	48,86	11,14	3,60	18,57
MEAN	**3,10**	**21,67**	**38,33**	**14,78**	**3,88**	**36,11**

Per ciascuna strategia, l'aumento del rapporto sender/consumer si riflette anche su un peggioramento delle prestazioni per via dell'aumento della pressione; tuttavia, non si trascorre mai più della metà del tempo in situazione di congestione, dimostrando l'efficacia del producer. Con un numero superiore di sender, crescerà anche il numero medio di messaggi in coda, mantenendosi sempre sotto la soglia di congestione. Si evince anche un maggior numero di volte in cui si verifica congestione non si traduce necessariamente in periodi molto lunghi di recovery: nel report di Aggregable si nota una diminuzione del CongestionTimes ma anche un aumento del tempo totale passato in quello stato, nonostante la crescita del rapporto sender/consumer. In entrambi i casi è evidente che la strategia di aggregazione restituisce una performance nel complesso migliore, con il 36% del TCT contro il 27% del drop, un numero medio di messaggi e congestioni lievemente inferiore. Ciò è dovuto alla possibilità di svuotare il buffer più efficientemente in situazioni di stress, ricorrendo alla sintesi, e non all'invio di più messaggi. A supporto di questa conclusione, si osservi cosa accade in corrispondenza di un rapporto mittenti/destinatari ancora più alto.

DROP 15 senders / 5 consumers

N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	6	38,71	21,29	23,52	5,87	64,52
2	5	37,47	22,54	18,24	5,77	62,44
3	6	42,60	17,40	26,35	7,02	70,99
4	6	36,52	23,48	20,30	5,47	60,87
5	8	36,33	23,68	15,20	5,08	60,54
6	5	39,38	20,62	27,28	6,77	65,64
7	5	44,56	15,44	32,40	6,98	74,27
8	8	33,47	26,53	10,17	4,07	55,79
9	6	40,52	19,49	24,27	6,07	67,53
10	5	46,65	13,35	34,49	9,13	77,75
MEAN	**6,00**	**39,62**	**20,38**	**23,22**	**6,22**	**66,03**

AGGREGABLE 15 senders / 5 consumers

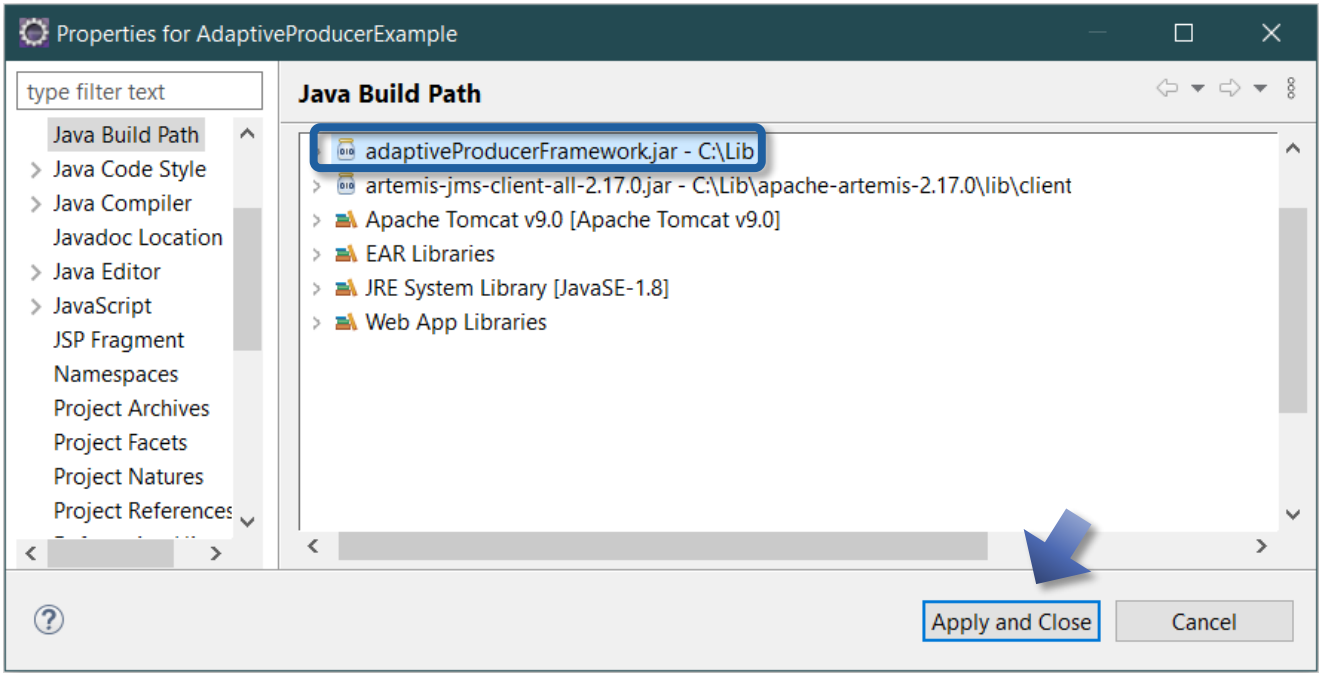
N	CT	TCT (s)	TNT(s)	LCP (s)	MQM	TCT/ST (%)
1	1	22,32	37,68	22,32	4,77	37,19
2	1	21,32	38,68	21,32	4,79	35,53
3	1	29,32	30,68	29,32	5,83	48,87
4	1	30,36	29,64	30,36	5,90	50,59
5	2	27,35	32,65	24,32	5,03	45,59
6	2	19,24	40,76	16,21	4,67	32,07
7	6	22,00	38,00	7,10	4,37	36,67
8	1	27,26	32,74	27,26	6,40	45,44
9	2	24,26	35,74	21,23	5,93	40,44
10	5	27,31	32,69	13,17	5,33	45,51
MEAN	**2,20**	**25,07**	**34,93**	**21,26**	**5,30**	**41,79**

Le simulazioni a rapporto maggiore determinano un generale peggioramento delle prestazioni in entrambi i casi; infatti, il numero dei messaggi in coda che si mantiene nell'intorno superiore della soglia di congestione. Tuttavia, la strategia di aggregazione riesce comunque a garantire un tempo medio di congestione accettabile, passandovi il 42% del totale.

LICENZA E DISTRIBUZIONE

La licenza scelta per il progetto è la GPL v.3, riportata nel root. È possibile trovarlo su GitHub con la sua release al seguente indirizzo: https://github.com/assuntaDC/AdaptiveProducerFramework.git. I package relativi al testing non sono compresi nella release.

Per utilizzare le soluzioni già implementate o estendere il framework è sufficiente scaricare il jar e aggiungerlo tra le dipendenze del progetto, quindi importare tutte le classi necessarie.



Si osservi un esempio minimale per l'invocazione delle operazioni principali.

```java
import adaptiveProducerFramework.producers.JMSProducerCreator;
import adaptiveProducerFramework.producers.Producer;
import adaptiveProducerFramework.producers.Sample;
import adaptiveProducerFramework.producers.exceptions.InvalidSampleTTLException;

public class SimpleTest {

    public static void main(String[] args) throws InvalidSampleTTLException {
        String acceptorAddress = "tcp://localhost:61616";
        String queueName = "testQueue";
        //Initialize client
        Producer producer = new JMSProducerCreator().createProducer(queueName, acceptorAddress);

        //start connection and queue monitoring
        producer.startProducer();

        //Send a sample
        Sample sample = new Sample(20.0, 500);
        producer.trySending(sample);

        //Close connection and queue monitoring
        producer.stopProducer();
    }
}
```

Segue un impiego in un contesto più strutturato. NodeDriver simula un sensore che genera campioni di temperatura e utilizza un oggetto producer per inviarli con una frequenza stabilita.

Per l'esempio completo visitare: https://github.com/assuntaDC/AdaptiveProducerExample.git.

```java
public class NodeDriver{

    private Producer producer;
    public long SENDER_PERIOD = 500;
    double max = 30;
    double min = 20;
    double range = max - min + 1;
    private ScheduledExecutorService executor;
    private ScheduledFuture<?> future;

    public NodeDriver(String queueName, String acceptorAddress, int id){
        producer = new JMSProducerCreator().createProducer(queueName, acceptorAddress);
    }

    public void startSending() {
        producer.startProducer();
        executor = Executors.newSingleThreadScheduledExecutor();
        future = executor.scheduleWithFixedDelay(new SendThread(), 0, SENDER_PERIOD, TimeUnit.MILLISECONDS);
    }

    public void stopSending() {
        producer.stopProducer();
        future.cancel(false);
        executor.shutdown();
    }

    private class SendThread implements Runnable{
        public void run() {
            int temperature = (int) ((int)(Math.random() * range) + min);
            try {
                Sample sample = new Sample(temperature, 500);
                producer.trySending(sample);
            }catch(InvalidSampleTTLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Riferimenti

ActiveMQ Apache. (n.d.). *Management*. Retrieved from Apache ActiveMQ Artemis User Manual: https://activemq.apache.org/components/artemis/documentation/latest/management.html

Jolokia. (n.d.). *Chapter 6. Jolokia Protocol*. Retrieved from Jolokia: https://jolokia.org/reference/html/protocol.html

RedHat. (n.d.). *CHAPTER 5. MONITORING BROKER RUNTIME DATA USING PROMETHEUS*. Retrieved from RedHat Customer Portal: https://access.redhat.com/documentation/en-us/red_hat_amq/7.4/html/managing_amq_broker/prometheus-plugin-managing

Villegas, N. M. (2017). Managing Trade-Offs in Adaptable Software Architectures. Elsevier.