Универзитет у Новом Саду

Факултет техничких наука, Нови Сад

Имплементација RabbitMQ концепата за размену порука коришћењем Open MPI стандарда

Рачунарски системи високих перформанси

Аутор:
Марија Петровић, E2 88/2022

Нови Сад, јануар 2023.

Сажетак

Тема овог рада се бави паралелизацијом реда порука налик на систем који се јавља код познатог софтвера RabbitMQ. Главни проблем јесте проблем познатији као ***bounded-buffer problem*** и присутан је код сличних механизама заснованим на произвођач-потрошач (*producer-consumer*) логици. Генерално посматрано, овај изазов јесте један о основних проблема синхронизације – оба процеса (*producer, consumer*) између себе имају један дељени ред/бафер који покушавају да користе у исто време, један процес да упише податке, а други да их преузме. Јасно је да до компликације долази јер један ресурс (ред порука у овом случају) истовремено бива манипулисан од стране два процеса. Овај рад се бави анализом и имплементацијом поменутог реда података коришћењем Open MPI (*Message Passing Interface*) стандарда.

Садржај

1. Увод

Још од ранијег периода развоја софтвера, један од изазова синхронизације рада више процеса истовремено, јесте претходно поменути проблем произвођача и потрошача. С једне стране архитектуре се налази процес који представља произвођача (*producer*), док се са друге стране може наћи један или више процеса названих потрошачима (*consumers*). Између ове две компоненте је присутан бафер одређеног капацитета који они међусобно деле, и користе га као ред (*queue*) како би се подаци смештали у њега и слали с једне стране на другу.

Задатак произвођача је да изгенерише податке, пошаље их у бафер и да даље настави са генерисањем, док потрошач преузима податке из реда и над њима врши акције у складу са даљом логиком система у оквиру којег је имплементиран. На основу овога, могуће је уочити потенцијалне препреке у описаном механизму:

- Произвођач би требао да генерише податке само када бафер није скроз попуњен. Ако је ред пун, произвођачу није дозвољено да уписује нове податке у њега.
- Потрошач(и) могу да читају податке из реда само ако он није празан. Ако у меморији нема ништа, преузимање не би требало да буде дозвољено или ће у супротном доћи до грешке.
- Произвођач и потрошач(и) не би смели да приступају дељеном баферу у исто време, односно у једном тренутку само један процес може да му приступи и да прави измене над њим.

За демонстрацију могућег решења детаљније ће бити разматран случај реда порука какав се среће код RabbitMQ софтвера. Решење ће бити реализовано употребом Open MPI стандарда који се бави паралелизацијом на нивоу процеса и има различите имплементације. У овом случају, као алат ће се користити програмски језик *Python* који у оквиру библиотеке *mpi4py* [1] нуди адекватну имплементацију поменутог стандарда.

2. Опис RabbitMQ технологије

RabbitMQ [2] је један од софтвера за размену порука њиховим складиштењем у ред (*message queue*), а познат је и по називу *message broker*. Он имплементира AMQP (*Advanced Message Queuing Protocol*) протокол, и може да подржи много шири спектар протокола тако да је у стању да испуни захтеве наметнуте од стране дистрибуираних окружења у којима се користи, као што су на пример висока доступност и скалабилност. Просто речено, у питању је програм у оквиру којег се дефинишу редови преко којих се врши трансфер порука између апликација повезаних на њих. Садржај поруке може бити информација било које врсте, главна идеја је да се оне привремено чувају у реду како би одредишне апликације могле да их преузму и даље процесирају.

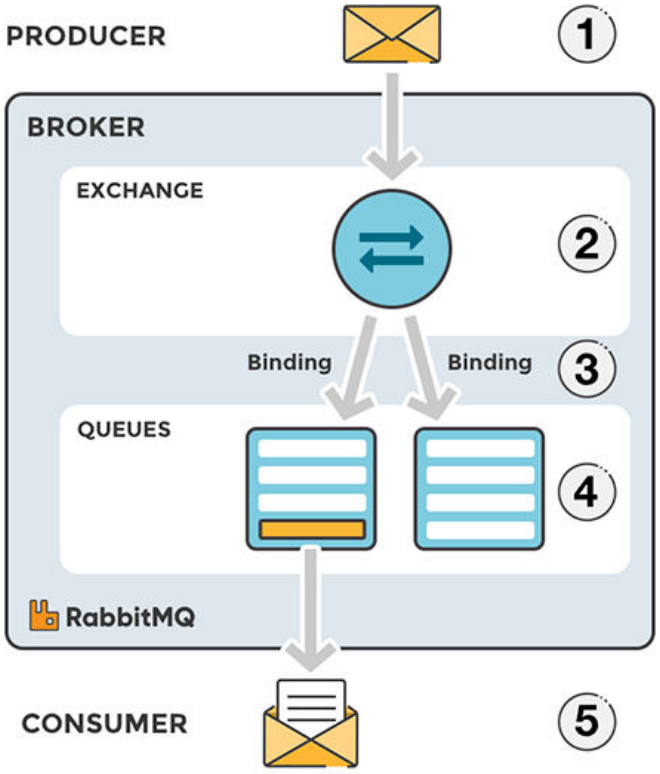
2.1. Циклус размене порука

Сама архитектура на којој почивају редови порука је прилично једноставна – постоје клијентске апликације (*producers*) чија је улога да креирају поруке и пошаљу их брокеру (што је у суштини ред порука), и затим друге апликације (*consumers*) ступају у комуникацију са редом и преузимају поруке намењене њима.

Међутим, пошто је сасвим реална ситуација да ће у неком систему бити неопходно више од једног реда, потребно је увести механизам који би обављао распоређивање порука. Када се заврши слање, RabbitMQ неће поруке директно сместити у ред, већ се оне пре тога шаљу другој компоненти која се зове *exchange*. Она је задужена за рутирање порука различитим редовима. Увођењем *exchange* система, ток би изгледао на следећи начин:

1. Произвођач (*producer*) шаље поруку *exchange* компоненти.
2. Exchange прима поруку и узима њене атрибуте – тип, кључ за рутирање и слично.
3. Прослеђује поруку у одговарајући ред на основу кључа (може се десити да је порука намењена и за више редова).
4. Порука стиже и смешта се у ред. Она ће се чувати ту све док је потрошач (*consumer*) не преузме.
5. Потрошач преузима поруку и даље наставља са адекватним процесирањем.

На слици 1 је дат илустративан пример.



Слика 1. Ток слања порука у RabbitMQ систему

2.2. Типови рутирања порука

Као што је већ поменуто, за рутирање порука је задужена *exchange* компонента. На основу различитих параметара, RabbitMQ подржава четири основна типа за рутирање [3]:

- директна размена (*Direct Exchange*),
- размена на основу топика (*Topic Exchange*),
- *Fanout Exchange* и
- размена на основу заглавља (*Headers Exchange*).

Директна размена (*direct exchange*) врши распоређивање порука на основу кључа за рутирање – *routing key*. Кључ представља један од атрибута поруке који ће произвођач да дода пре него што је пошаље. Другим речима, он може да се посматра и као "адреса" на коју ће се порука даље проследити. Сваки ред поседује кључ за повезивање који одговара кључу за рутирање, тако да ће се приликом испоручивања порука проверити да ли се та два атрибута поклапају.

Размена на основу топика ради на принципу слања порука у ред(ове) у зависности од тога који образац задовољава кључ за рутирање (*wildcard pattern matching*). Приликом дефинисања кључа, речи које улазе у назив морају бити одвојене тачком. Потрошач жели да нагласи који топици су му од интереса – он ће дефинисати образац за свој ред и повезаће га са *exchange* компонентом. Све поруке чији се кључеви за рутирање поклапају са шаблоном биће прослеђене у ред.

Fanout Exchange ће поруку реплицирати у онолико копија колико има редова, и затим ће сваком присутном реду доделити једну од њих. У овом случају вредност кључа за рутирања се уопште не посматра, јер се порука прослеђује свима. Овај приступ је препоручљиво користити ако су захтеви система такви да је једну поруку потребно послати већем броју потрошача како би се порука обрадила на различите начине.

И на крају, размена на основу заглавља функционише слично као и на основу топика, али с тим да се не посматрају кључеви за рутирање већ вредности уписане у заглавље поруке. Приликом повезивања реда са *exchange* компонентом специфицира се аргумент *x-match* који може да има вредности *any* или *all*, и говори да ли све вредности атрибута заглавља морају бити задовољене или само нека од њих.

У овом раду ће бити имплементиран само један од начина – директна размена.

3. Имплементација

У оквиру овог поглавља биће представљено поједностављено решење како би се уз коришћење *OpenMPI* стандарда могла имплементирати RabbitMQ директна размена порука између произвођача и потрошача. У конкретном примеру биће посматран случај када је у систему присутан један произвођач (*producer*), док се са друге стране налазе два потрошача (*consumers*), сваки са својим редом порука. Поруке за слање ће се специфицирати приликом корисничког уноса.

3.1. Демонстрација решења

Ток размене порука је симулиран у оквиру једног глобалног *OpenMPI* комуникатора, који у ствари представља групу процеса, где је замишљено да сваки од процеса представља једну од компоненти система. Пошто се процеси у комуникатору разликују на основу ранка, извршена је следећа подела:

- producer процес (`rank = 0`),
- exchange процес (`rank = 1`),
- consumer_1 процес (`rank = 2`) и
- consumer_2 процес (`rank = 3`).

Покретање програма са ова четири процеса се врши на следећи начин:

```
mpiexec -n 4 python3 main.py
```

Пример 1. Команда за покретање *python* скрипте са четири процеса

За репрезентацију редова порука је употребљена *Python* класа `Queue`, која ради на FIFO (*First-In, First-Out*) принципу. Она припада `queue` модулу, специјално оптимизованом за програмирање у вишенитном окружењу, и нарочито се користи када је потребно имплементирати безбедну размену информација између нити.

Поруке се задају корисничким уносом у оквиру процеса произвођача, и приликом тога наводе се два атрибута – садржај поруке и вредност кључа за рутирање. Након тога, спакована порука се прослеђује процесу задуженом за симулацију компоненте за расподелу порука. Овде кључну улогу играју *OpenMPI* методе за слање, односно примање порука – `comm.isend`, `comm.irecv`. Ове методе су неблокирајуће, што значи да процес који иницира слање или примање поруке неће сачекати да се заврши операција, већ ће наставити са другим послом. У већини система где постоје захтеви за сталном комуникацијом и разменом података до којих се долази помоћу неког

израчунавања, перформансе могу значајно да се повећају јер ће се две аутономне операције извршавати

истовремено и независно. У овом случају употребљена је и функција **request.wait** која ће да тестира да ли је захтев извршен или није. У наставку је дат пример изворног кода процеса задуженог за рутирање порука.

```
    # direct exchange
    if rank == 1:
        while True:
            try:
                req = comm.irecv(source=0, tag=11)
                message = req.wait()
                if message['routing_key'] == 'q1':
                    req = comm.isend(message, dest=2, tag=11)
                    req.wait()
                elif message['routing_key'] == 'q2':
                    req = comm.isend(message, dest=3, tag=11)
                    req.wait()
            except MPI.Exception as e:
                print("Exchange failed to receive or send message to
consumer(s):", e.error_string)
```

Пример 2. Део изворног кода *exchange* процеса

Када *exchange* процес прими поруку његов задатак је да изврши проверу вредности атрибута кључа, и да на основу њега поруку даље проследи одговарајућем процесу. Ако је кључ *q1*, порука ће се проследити процесу са ранком 2, док ако је у питању *q2*, порука се упућује процесу ранка 3. Поруке чији се кључ за рутирање не поклапа са ознакама присутних редова неће бити обрађене.

Даље, процеси 2 и 3 прихватају поруке њима додељене, и смештају их у редове за које су задужени. То је реализовано на једноставан начин, позивом функције **put_nowait** из **Queue** класе (пример 3) која ће убацивати поруке у ред без блокирања.

```
req = comm.irecv(source=1, tag=11)
received_message = req.wait()
queue1.put_nowait(received_message)
data = read_json("queue1.json")
data.append(received_message)
save_to_json_for_consumer("queue1.json", data)
```

Пример 3. Додавање поруке у ред порука

Потрошачи ће затим да преузимају поруке из реда помоћу **get_nowait**, која је такође неблокирајућа функција. У том случају, ако се покуша избацивање поруке из празног реда, да не би

дошло до грешке, потребно је сваки пут проверити статус реда за шта су такође обезбеђене методе класе.

У примеру изнад се може приметити да је уведена и перзистенција редова порука у *.json* фајлове, како би се отклонила могућност губљења неке информације.

4. Закључак

Проблем произвођача и потрошача, описан на почетку, је раније представљао много већу бригу приликом развоја комплексних система који су укључивали комуникацију више различитих процеса. Временом су настале многе асинхроне платформе за размену порука, тако да би комуницирали, произвођач и потрошач нису у обавези да буду повезани у исто време. То значи да је произвођач у стању да пошаље поруку потрошачу иако је он тренутно недоступан – порука ће му бити испоручена када он буде поново био у могућности за то. Такође, свакако је потребно да се раздвоје логике потрошача и произвођача (*decoupling*) како би се обе компоненте могле скалирати независно. Додатно, то пружа и већу отпорност на отказе, јер када су раздвојени елементи, пад једног неће утицати на прекид рада другог елемента архитектуре.

Све набројане особине одликују RabbitMQ – произвођач и потрошач су потпуно независни један од другог и могуће их је одвојено скалирати, док се између њих налази један или више редова порука преко којих комуницирају.

Решење изложено у оквиру овог рада представља симулацију рада основних концепата присутних у RabbitMQ имплементацији, посматраних на вишем нивоу апстракције. Захваљујући *OpenMPI* подршци за размену порука на различитим паралелним архитектурама, омогућена је комуникација између процеса који у овом случају представљају одвојене компоненте система. Тренутна имплементација има простора за даљи напредак са аспеката логике и функционалности, што је и пожељно, како би функционисање било што приближније оригиналном. На пример, као проширење би се могло разматрати увођење механизма за поновно слање порука у случају да се догодила нека сметња приликом првог покушаја. На тај начин би се систем осигурао да неће доћи до губитка података у транзицији. Исто тако, могуће је увести и алармирање у случају да се примети необично понашање које би евентуално могло да проузрокује веће проблеме (нпр. ако се уочи да потрошач не преузима поруке онолико брзо колико иначе то ради, или ако произвођач шаље поруке неуобичајено брже него што друга страна стиже да их обради), како би се на време реаговало и спречила потенцијална катастрофа.

5. Литература

[1] https://mpi4py.readthedocs.io/en/stable/overview.html

[2] https://www.rabbitmq.com/tutorials/amqp-concepts.html

[3] https://hevodata.com/learn/rabbitmq-exchange-type/