GROUPCAST

Total Order Multicast and Group Communication

Josep Sugrañes Riera
Sergi Canet Vela

INDEX

1. Especificacions del projecte

Aquesta pràctica es basa en implementar un Group Communication que proporcioni Total Order Multicast (TOM) a un nombre de participants fent ús de les funcions de la llibreria proporcionada PyActor treballada a classe (https://github.com/pedrotgn/pyactor). S'implementaran dos tipus de TOM, la primera basada en Sequencers i la segona en Lamport Clocks (la qual en el nostre cas, no la hem realitzat).

El Group Communication implementa un Group Membership Service (GMS), el qual s'encarrega de mantenir el rastreig dels usuaris participant al grup. Ha de contenir els mètodes: join, leave i get_members. Per a la implementació, es pot aprofitar el Tracker de la pràctica 1, però serà necessari incloure fault-tolerance per a aquells membres que fallin sense cridar explícitament al mètode leave.

En el TOM, cada membre del grup ha entregar tots els missatges enviats al grup en ordre lineal, la qual cosa no implica que sigui necessari fer-ho en temps real. Molts protocols TOM encuen els missatges rebuts de manera no ordenada a cada membre fins que poden ser entregats a l'aplicació en l'ordre correcte. Aquesta cua (inicialment buida), s'utilitza posteriorment per reordenar els missatges de forma local abans d'enviar-los a la aplicació. Ha de contenir els mètodes:

- multicast: aquest mètode serà usat per cada membre per enviar un missatge a la resta de membres del grup.
- receive: un membre fa una crida a aquest mètode per transmetre un missatge a un altre membre.
- process_msg: aquest mètode serà cridat per un membre només quan sigui segur enviar el missatge a la aplicació per a processar-lo (tenint en compte que ha de ser ordre lineal).

En el cas del TOM amb sequencer, un sequencer és un procés que assigna un únic número de seqüència (timestamp) a cada missatge que reb, i es fa multicast a tots els altres membres del grup. Per a la implementació, es pot aprofitar la classe Peer de la pràctica 1, però serà necessari que el propi sequencer no faci multicast dels missatges. El multicast serà realitzat per tots els membres del grup: el mètode multicast enviarà un missatge a tots els membres amb el número de seqüència únic que proporcioni el sequencer. Per tant, els missatges seran enviats en l'ordre correcte al mètode process_msg. Els missatges rebuts en ordre no lineal es seran encuats en un buffer fins a que el missatge amb el número de seqüència esperat arribi. Com que el seqüencer té un single point of failure (SPOF), també és necessari incloure un mecanisme de failure tolerance, com, per exemple, fer ús de l'algoritme bully leader election per escollir un nou sequencer en cas de fallada.

1.1. Requeriments específics

a) S'ha de crear un grup de n peers que entrin al group i comencin a fer multicast dels missatges, on n sigui un paràmetre que pugui cambiar en cada simulació.
b) Comprovar que tots els peers reben els events en el mateix ordre.
c) Abans del TOM, els n peers han d'unir-se primer al group. A la fase del multicast, s'assumirà que no els peers no poden marxar ells mateixos cridant al mètode leave.
d) Per a forçar el reordenament dels missatges multicast, s'haurà d'introduir un retard fent ús de la funció sleep.
e) És obligatori que les implementacions treballin en dos terminals separats.

1.2. Tasques a realitzar i mètode d'avaluació

a) Group Membership Service (GMS) (30%). Implementació del GMS. Incloure els tests apropiats per comprovar el correcte funcionament.
b) Total Order Multicast (TOM) amb sequencer (30%). Implementació del TOM amb sequencer. Inclou la implementació del algoritme bully leader election.
c) Total Order Multicast (TOM) amb Lamport Clocks (15%). Implementació del TOM amb lamport clocks. (en el nostre cas, aquesta part no la hem realitzat)
d) Preguntes teòriques (20%).
e) Repositori github (5%). Crear un repositori personal amb el codi i la documentació. És recomanable incloure proves de code coverage i unitary tests.

2. Disseny

Hem decidit crear dues classes: Grup i Peer (de la qual en tenim dues versions: una per a la execució en un únic terminal i una altra per a la execució en diferents consoles). Addicionalment tenim tres classes més: Printer (per imprimir per pantalla), Main (on es fan les proves pel cas de la execució en el mateix terminal).

Hem plantejat la nostra solució de la següent manera: un peer únicament es comunica amb el grup quan vol entrar per a saber els membres que hi ha en el grup i a partir d'aquell moment el propi peer avisa a tots els altres membres per a que l'afegeixin a la seva llista de membres.

El grup però, controla que els peers facin un announce cada 10 segons, però això servirà únicament per actualitzar la llista que té el grup per si entra un nou peer, ja que la aplicació es descentralitzada.

Els peers tenen un diccionari (cache) de tots els altres peers, on la clau és la URL d'aquell peer i el valor és el proxy. D'aquesta manera ens estalviem fer lookup_url's periòdicament que ens poden donar timeouts.

No podem saber si un peer ha caigut amb certesa perquè si preguntéssim al grup si aquest ha fet l'announce, estaríem centralitzant l'aplicació, ja que dependríem del grup. Per tant, per a fer el multicast dels missatges utilitzem el Future de la llibreria PyActor. Podríem eliminar el peer en cas que hagués caigut, però necessitaríem que la llibreria PyActor permetés fer un timeout del

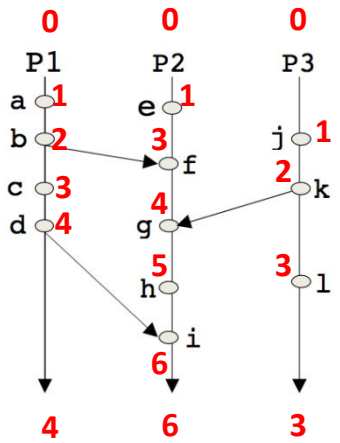
Future. D'aquesta manera, si no rebéssim resposta al cap d'un cert temps (timeout), podríem suposar que el peer ha caigut i eliminar-lo de la llista de membres de cada peer.

El sequencer serà, inicialment, el primer peer que entra dins al grup. A partir d'aquest moment, quant entra un nou peer al grup, pregunta a un altre membre qualsevol qui és el sequencer per a poder tenir la referencia a aquest.

Després de tres intents fallits que els peers demanin la prioritat al sequencer, considerem que el sequencer ha caigut i s'aplica el mètode Bully per a triar-ne un de nou.

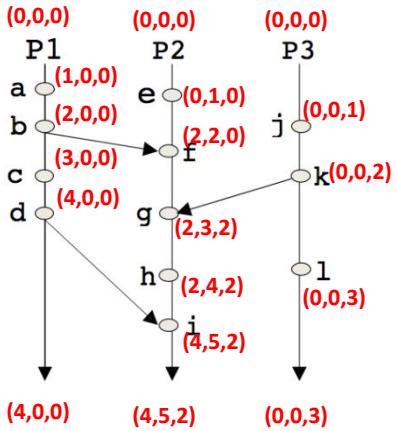
3. Preguntes teòriques

a) Lamport va definir una idea de temps lògic basat en ordenar esdeveniments – la anomenada relació "happened-before". Demostra com un ordenació lineal global d'esdeveniments pot ser aconseguida basant-nos en aquesta idea dels rellotges lògics de Lamport per a cada event de la figura següent:



Amb els rellotges lògics de Lamport podem ordenar events que tenen relació, és a dir podem saber si un event "a" ha ocorregut abans que "b" (si a i b són events del mateix procés o bé si a és l'event d'un missatge enviat a un altre procés i b és l'event del missatge rebut per l'altre procés) i demostrar, per tant, que a→b.

b) Com els vector-clocks extenen aquesta idea dels rellotges lògics de Lamport?



En alguns casos no només ens pot interessar ordenar elements que tenen relació sinó que volem tenir-ho totalment ordenat i ens interessa també conèixer la hora actual (per exemple, per sistemes de temps real). En aquest cas, ens interessa tenir rellotges externs físics. Amb els vector clocks, podem retrocedir i mirar quin event succeeix abans en el temps.

c) Justifica el problema d'elecció del líder i digues el cost de comunicació per a l'algoritme bully.

Assumint només crash failures, quin és el mínim nombre de participants necessaris per a que la elecció sigui exitosa?

Necessitem que un membre controli el sequencer de manera que es proporcionin els números de seqüència correctes per als missatges (aquest membre és el líder), però podria donar-se el cas que aquest membre se'n vagi del grup o es quedi bloquejat. Seguim necessitant que es pugi enviar i rebre missatges al grup i aquests es mostrin ordenats. Per això necessitem escollir un altre líder que passi a controlar el sequencer.

L'algorisme bully leader election s'encarrega de, quan un peer detecta que el líder s'ha caigut, notificar als peers amb ID més gran que el seu. Els processos que estiguin funcionant correctament, retornaran 'OK' a aquest peer. A partir d'aquí, aquests procesos que han retornat 'OK' tornaran a notificar als processos amb ID més gran fins que només es retorni 'OK' per un sol peer, que serà el que tenia l'ID més gran. Ara aquest peer serà el nou líder.

D'aquesta manera, és possible que caigui el líder i seguirà funcionant el grup.

El cost serà més gran a mesura que tinguem més peers, ja que s'hauran d'enviar més missatges per a poder escollir un nou líder.

Es necessari que hi hagi, com a mínim, dos participants al grup, el líder que falla i un altre peer que en el moment que se'n adona esdevé nou líder perquè no hi ha cap altre peer amb id més gran.

d) Compara els dos algoritmes de comunicació pel que fa al cost de comunicació, fault-tolerance i escalabilitat.

Pel que fa al cost de comunicació, és millor el TOM amb sequencer, ja que s'envien molts menys missatges que amb Lamport (necessitem que s'enviin a tothom els ACK's) i, pel que fa al fault-tolerance tenim que també és més eficient la implementació amb sequencer, perquè en cas que falli el peer que el controla, es poden fer eleccions. En canvi, en el cas del Lamport, només que falli un peer els missatges es descarten perquè no s'obtenen tots els ACK's.
En quant a la escalabilitat, és molt millor utilitzar Lamport, ja que no saturem un únic peer com passa amb l'altra implementació (saturem el peer que controla el sequencer).

4. Link GitHub

- https://github.com/josep0704/GroupCast