GroupCast

Total Order Multicast and Group Communication

JORDI BORRELL VIVES-MIHAI SERBAN COPIL

SISTEMES DISTRIBUÏTS

2016-2017

Introducció

L'objectiu de la segona tasca és implementar una biblioteca de comunicació de grup minimalista que proporciona Total Order Multicast (TOM) a un nombre de participants a través de la biblioteca PyActor. TOM és una primitiva molt bàsica dels sistemes distribuïts. Per exemple, un grup d'arxius replicats no pot estar en el mateix estat llevat que totes les rèpliques apliquen les actualitzacions en el mateix ordre. En aquests tipus de situacions és on es fa ús de TOM.
En aquesta tasca, anem a comparar dues implementacions diferents de TOM. La primera aplicació es basa en l'ús de seqüenciadors i la segona implementació es tractara d'un sistema que faci us del teorema de Lamport(Lamport Clocks).

Decisions de disseny

Group:

Per a comprovar que cap dels peers no falla i no ha marxat del group utilitzem la mateixa implementació que en el BitTorrent, obligant a que els peers vagin anunciant-se al group i que aquest elimini als peers que porten massa estona sense anunciar-se.

Al group tindrem un diccionari amb les url dels peers com a clau, i la data de l'últim cop que s'han anunciat com a valor per a retornar totes les claus en el getMembers i poder treballar amb tots els peers del grup i a la vegada poder comprovar que aquests es vagin anunciant i cap hagi marxat sense fer un leave; un altre diccionari amb la id dels peers coma clau i la seva url com a valor per a poder realitzar les votacions i elegir un nou sequencer en cas de que l'actual caigui o marxi; un enter que serà el que utilitzara el group per anar donant les ids als peers de manera que no se repeteixin i un booleà per a què tan sols un peer pugui estar fent la votació a la vegada.

Peers:

En els peers tenim en cadascun les variables data, que és una llista on guardaran els missatges processats; una PriorityQueue falten on guardaran els missatges rebuts però no processats perquè no tenen l'ordre que estan esperant; un enter nextMessage que serà l'ordre del següent missatge que esperen rebre; un enter timeStamp que tan sols usara el sequencer per a donar l'ordre corresponent als peers que li demanin per a què tots els missatges estiguin ordenats; un enter nId que serà la id del peer; un altra variable url que serà la seva url; i una última variable seqId que serà la id del sequencer, que s'utilitzarà en cas de que l'actual sequencer caigui i s'estigui fent una votació per elegir-ne un de nou.

En el mètode multicast, en el cas de que el sequencer no doni el timeStamp i salti un TimeoutError s'iniciarà la votació d'un nou sequencer i no s'enviarà el missatge que s'estava intentant enviar. Si no es fa la votació, i per tant el sequencer ha donat el timeStamp corresponent s'envia el missatge a tots els peers juntament amb aquest ordre.

Per realitzar la votació d'un nou sequencer s'agafa la llista de totes les urls dels peers del grup amb les seves ids i s'ordenen per ids de major a menor, de manera que es va mirant peer a peer, començant pel que té la id més alta, que serà l'últim peer que s'ha unit al group. El primer peer que contesti, o si s'arriba al peer que està duent a terme la votació, serà el nou sequencer, de manera que el peer actiu amb la id més alta serà l'elegit. Un cop el peer que està fent la votació sap qui serà el nou sequencer ho comunica a tota la resta de peers i acaba.

A l'hora de rebre un missatge, si aquest té l'ordre del següent missatge que s'espera processar, aquest es processa, sinó s'envia a la cua dels missatges que falten per processar.

Un cop un missatge s'esta processant, aquest es col·loca a la llista de missatges processats, s'incrementa el nombre de l'ordre del següent missatge que s'espera processar i es mira la cua de missatges per processar, si no està buida i el missatge amb l'ordre més baix que hi ha és l'ordre del següent missatge que esperem processar aquest es processa recursivament, fins que l'ordre del missatge que tenim pendent de processar amb el nombre de ordre més baix no correspon a l'ordre del següent missatge que esperem processar o la llista de missatges pendents de processar es quedi buida.

Joc de Proves

A part d'un main en els peers per fer en aquests totes les proves adients, s'han fet els següents jocs de proves per testejar els següents requisits de la pràctica.

En aquestes proves donem per suposat que almenys hi haurà el sequencer i dos peers més i que no s'usarà el main dels peers.

Per executar aquests tests hem de iniciar primer el group en un terminal, després iniciar els peers amb el nombre de peers que vulguem (mínim 2) en un altre terminal, i finalment iniciar el test que vulguem en un tercer terminal, la informació dels peers del group ens sortirà per el terminal del group, la informació del que s'està fent sortirà pel terminal del test que estem executant.

Provide here a fault-tolerance mechanism to detect peers that fail without explicitly calling the leave method:

```
from pyactor.context import set_context, create_host, sleep, shutdown
from peer import *
from group import *


if __name__ == "__main__":
        set_context()
        url = 'http://127.0.0.1:6971/'
        host = create_host(url)

        g = host.lookup_url('http://127.0.0.1:6969/group', Group, future=True)
        g = g.result()

        seq = host.lookup_url('http://127.0.0.1:6970/seq', Peer, future=True)
        seq = seq.result()

        p1 = host.lookup_url('http://127.0.0.1:6970/peer1', Peer, future=True)
        p1 = p1.result()

        p2 = host.lookup_url('http://127.0.0.1:6970/peer2', Peer, future=True)
        p2 = p2.result()

        print("Aqui tenim tots els peers")
        g.infopeer()
        sleep(3)
        print("Aturem el peer1")
        p1.stopPeer()
        sleep(3)
        g.infopeer()
        sleep(20)
        g.infopeer()
        sleep(10)

shutdown()
```

Amb aquest test podem veure com tenim la informació de tots els peers dintre del grup, i passats uns segons després d'aturar el peer1 aquest deixa de sortir a la informació del grup perquè s'ha detectat que deixa d'estar actiu i s'ha eliminat.

Total order multicast amb missatges enviats en ordre:

```
from pyactor.context import set_context, create_host, sleep, shutdown
from peer import *
from group import *


if __name__ == "__main__":
        set_context()
        url = 'http://127.0.0.1:6971/'
        host = create_host(url)

        g = host.lookup_url('http://127.0.0.1:6969/group', Group, future=True)
        g = g.result()

        seq = host.lookup_url('http://127.0.0.1:6970/seq', Peer, future=True)
        seq = seq.result()

        p1 = host.lookup_url('http://127.0.0.1:6970/peer1', Peer, future=True)
        p1 = p1.result()

        p2 = host.lookup_url('http://127.0.0.1:6970/peer2', Peer, future=True)
        p2 = p2.result()


        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola1")
        p1.multicast("hola1")
        sleep(3)
        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola2")
        p1.multicast("hola2")
        sleep(3)
        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola3")
        p1.multicast("hola3")
        sleep(3)
        g.infopeer()
        sleep(10)


shutdown()
```

Amb aquest test podem veure com enviant tres missatges diferents un després de l'altre els hi arriben a tots els peers amb l'ordre correcte.

Total order multicast amb missatges enviats desordenats:

```
from pyactor.context import set_context, create_host, sleep, shutdown
from peer import *
from group import *


if __name__ == "__main__":
        set_context()
        url = 'http://127.0.0.1:6971/'
        host = create_host(url)

        g = host.lookup_url('http://127.0.0.1:6969/group', Group, future=True)
        g = g.result()

        seq = host.lookup_url('http://127.0.0.1:6970/seq', Peer, future=True)
        seq = seq.result()

        p1 = host.lookup_url('http://127.0.0.1:6970/peer1', Peer, future=True)
        p1 = p1.result()

        p2 = host.lookup_url('http://127.0.0.1:6970/peer2', Peer, future=True)
        p2 = p2.result()


        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola1")
        p1.multicast("hola1")
        sleep(3)
        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola2 amb delay")
        p2.multicast2("hola2")
        sleep(3)
        g.infopeer()
        sleep(3)
        print("Enviem el missatge hola3")
        seq.multicast("hola3")
        sleep(3)
        print("Veiem que tan sols s'ha processat el primer missatge i tenim un pendent per processar, ja que el segon
missatge encara no ha arribat")
        g.infopeer()
        sleep(10)
        print("Veiem que en quan ha arribat el segon missatge ja es processen tots correctament i en ordre")
        g.infopeer()
        sleep(10)


shutdown()
```

Amb aquest test podem veure com enviant el segon missatge amb delay aquest no arriba als peers, així al enviar un tercer missatge sense que el segon els hi hagi arribat aquest tercer no es processa i tan aviat com els hi arriba el segon missatge processen aquest i l'altre que tenien guardat.

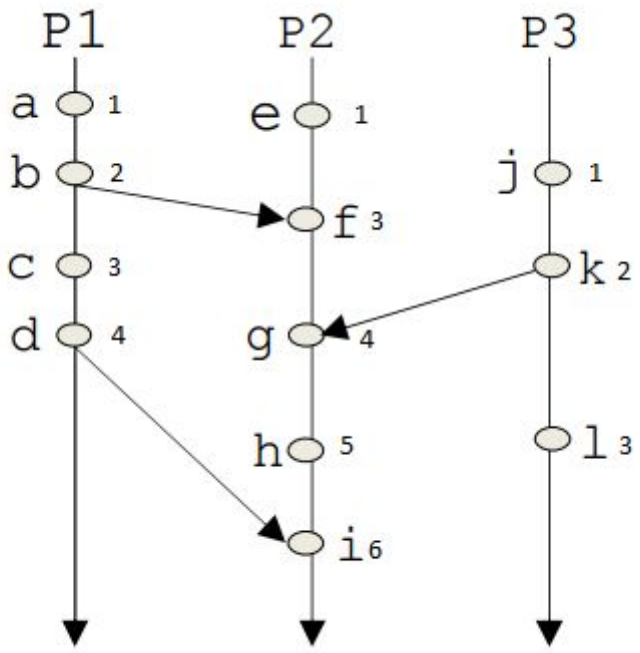
Total order multicast amb caiguda de sequencer:

```
from pyactor.context import set_context, create_host, sleep, shutdown
from peer import *
from group import *


if __name__ == "__main__":
        set_context()
        url = 'http://127.0.0.1:6971/'
        host = create_host(url)

        g = host.lookup_url('http://127.0.0.1:6969/group', Group, future=True)
        g = g.result()

        seq = host.lookup_url('http://127.0.0.1:6970/seq', Peer, future=True)
        seq = seq.result()

        p1 = host.lookup_url('http://127.0.0.1:6970/peer1', Peer, future=True)
        p1 = p1.result()

        p2 = host.lookup_url('http://127.0.0.1:6970/peer2', Peer, future=True)
        p2 = p2.result()


        g.infopeer()
        sleep(3)
        print("enviem el missatge hola1")
        p1.multicast("hola1")
        sleep(3)
        g.infopeer()
        sleep(3)
        print("Fem marxar al sequencer")
        seq.leave()
        sleep(3)
        print("Intentem enviar un missatge per iniciar la votacio d'un nou sequencer")
        p1.multicast("hola2")
        sleep(5)
        print("amb la votacio finalitzada enviem el missatge hola2 per provar que ha funcionat correctament")
        p2.multicast("hola2")
        sleep(3)
        g.infopeer()
        sleep(10)


shutdown()
```

En aquest test podrem veure com al fer que marxi el sequencer, al enviar el segon missatge, el peer no obté el timestamp del sequencer, de manera que inicia una votació, al finalitzar aquesta podem veure que al enviar un nou missatge aquest arriba correctament a tots els peers, així podem provar que hi ha un nou sequencer i tot segueix funcionant correctament.

Preguntes Teòriques

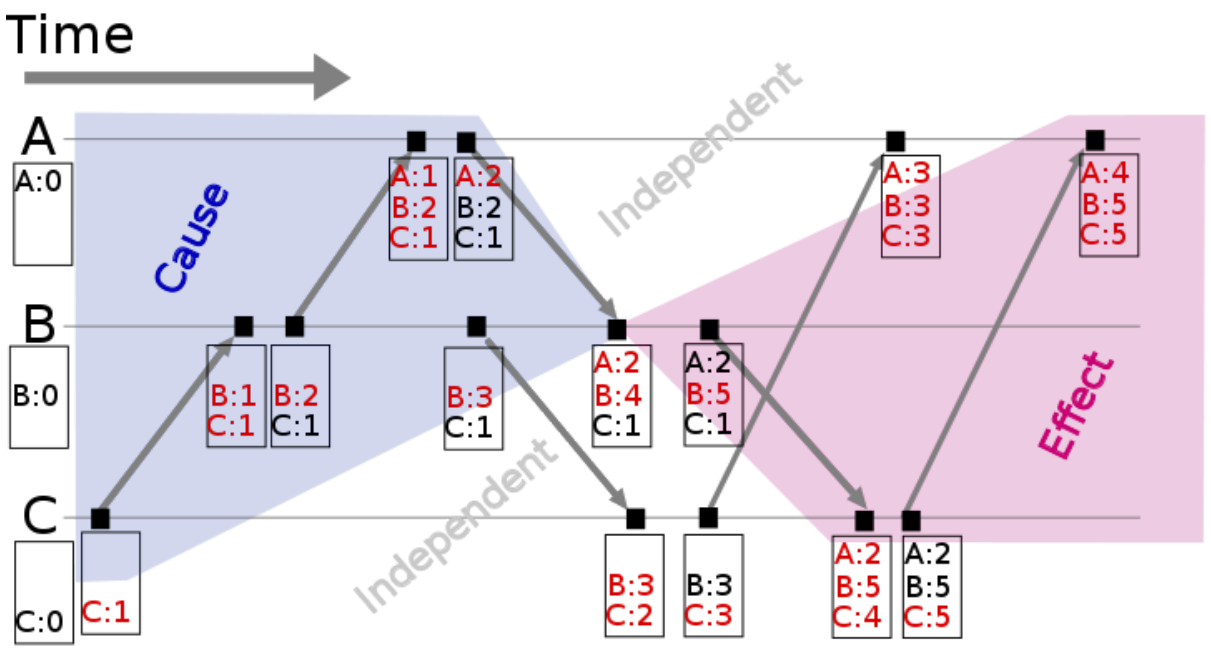
a. Lamport defined a notion of logical time based on event ordering – the so called “happenedbefore” relation. Show how a global total ordering of events can be achieved based on this notion of Lamport’s logical clock by specifying the Lamport clocks for each event in the next figure.



b. How do vector-clocks extend the notion of Lamport's logical clocks?

Un rellotge vectorial manté constància dels rellotges lògics de tots els processos involucrats amb l'us de vectors de dades(a dintre del vector es guarda un rellotge per cada procés). Cada vegada que experimenta un esdeveniment es van actualitzant els rellotges lògics de dintre del vector en consecuencia a l'esdeveniment que ha succeït seguint la següent pauta:

- Inicialment tots els rellotges están a zero.
- Cada vegada que un procés experimenta un esdeveniment intern, incrementa el seu propi rellotge lògic en el vector en 1.
- Cada vegada que un procés es prepara per enviar un missatge, s'envia tot el seva vector juntament amb el missatge que s'envia.
- Cada vegada que un procés rep un missatge, s'incrementa el seu propi rellotge lògic en el vector en 1 i s'actualitza cada element en el seu vector prenent el màxim del valor dels rellotges lògics continguts dintre del seu vector i el valor dels rellotges lògics continguts dintre del vector que rep.



c. State the leader election problem and tell the communication cost for the Bully algorithm. Assuming only crash failures, which is the minimum number of participants that are needed for an election to be successful?

Molt de programari en sistemes distribuïts requereixen un procés especial anomenat el coordinador o el líder. El coordinador és elegit entre els processos que participen un servei

distribuït en particular quan el servei comenci a funcionar o quan el coordinador deixa d'estar actiu.
El procediment de selecció del coordinador es l'elecció. El problema de l'elecció del líder és per a cada procés, finalment, decidir si es tracta d'un líder o no subjecte a una decisió pròpia.
Un dels algoritmes de les eleccions més populars és l'algoritme de bully. No obstant això, aquest algoritme és costós, ja que genera O missatges (n^2) per n processos(en el pitjo cas). També el temps de retard fins a la finalització de l'elecció és considerablement llarg.

El número exacte de missatges per determinar el coordinador es calcula amb la següent formula:

$$T_m = (N - P + 1)(N - P) + N - 1 \qquad (1)$$

where:
T_m – the number of exchanged messages between processes when process P detects failure of the coordinator,
N – the total number of processes,
P – the ID of the process that detects the failure of the coordinator.

Per lo tant en el millor cas serien O(n)=n-1 i en el pitjor cas O(n^2)=n^2-1

Per a que una elecció sigui exitosa es necessitarien 2 participants; ja que quan un participant nota la falta d'un lider, aquest comença la votació per elegir un nou lider segons la ID més alta involucrada. En un sol participant no tindria molt de sentit usar aquest tipus d'algorisme.

d. Compare both communication algorithms in terms of communication cost, fault-tolerance, and scalability.

Pel que fa a nivell de costos de comunicació, son força reduïts ja que es estableix la comunicació i s'envia el missatge sense esperar resposta de si aquest ha sigut o no rebut; en canvi en el lamport al enviar un missatge s'ha d'esperar per confirmar l'estat del missatge al receptor, per tant la comunicació es manté més temps i s'envien més paquets.
A nivell de tolerància a les fallades, el sequencer es un SPOF(single point of failure) ja que si es cau el sequencer es cau el sistema, encara que per solucionar aquest problema es pot implementar el algorisme de bully per la elecció d'un nou lider.
Per que fa a l'escalabilitat el sequencer es poc escalable ja que una gran quantitat de processos podrien sobrecarregar el sequencer disminuint la seva efectivitat.

Codi

Group:

```
from datetime import datetime
from sets import Set
from pyactor.context import interval
from pyactor.context import set_context, create_host, sleep, shutdown, serve_forever
from peer import *
from random import sample




class Group(object):
	_tell = ['announce', 'comprovarPeers', 'init_start', 'infopeer', 'esborrar', 'esborrarId', 'setVotant', 'afegirPeer']
	_ask = ['getMembers', 'getNId', 'getVotant', 'getPeerIds']
	_ref = ['announce', 'esborrar', 'getMembers', 'afegir_host']

	def __init__(self):
		self.tDictionary = {}
		self.nId = -1
		self.votant = False
		self.peerId = {}

	def init_start(self):
		self.interval_check = interval(self.host, 1, self.proxy, 'comprovarPeers')

	def afegirPeer(self, idPeer, urlPeer):
		try:
			self.peerId[idPeer] = urlPeer
		except:
			self.peerId = {idPeer:urlPeer}

	def announce(self, urlPeer):
		try:
			self.tDictionary[urlPeer] = datetime.now()
		except:
			self.tDictionary = {urlPeer:datetime.now()}

	def getVotant(self):
		return self.votant

	def setVotant(self, bolea):
		self.votant = bolea

	def esborrar(self, urlPeer):
		try:
			#if refPeer.timeStamp != -1:
			#	timeStamp = refPeer.timeStamp
			#	newSeq = choice(self.getMembers())
```

```
                #           newSeq.setTimeStamp(timeStamp)
                #           for peer in self.getMembers():
                #                   peer.afegir_sequencer(newSeq)

                del(self.tDictionary[urlPeer])
        except:
                None

def esborrarId(self, idPeer):
        try:

                del(self.peerId[idPeer])
        except:
                None

def getMembers(self):
        try:
                keys = self.tDictionary.keys()
                return keys
        except Exception as e:
        return []

def getPeerIds(self):
        return self.peerId

def comprovarPeers(self):
        tactual = datetime.now()

        for urlPeer, t_peer in self.tDictionary.items():
                result=tactual-t_peer

                if result.total_seconds() <= 20:
                        None
                else:
                        try:
                                peer = self.host.lookup_url(urlPeer, Peer)
                                print "eliminar "+urlPeer
                                self.esborrar(urlPeer)
                                self.esborrarId(peer.getId())
                                peer.stop_interval()
                                peer.stopPeer()
                        except TimeoutError as e:
                                None

def infopeer(self):
        print "////////////////////////////////////"
        for peerUrl in self.tDictionary.keys():
                try:
                        x = self.host.lookup_url(peerUrl, Peer)
                        print " -"+x.getId()+": "
                        print x.getMessage()
                        noProc = x.getFalten(future=True)
                        noProc = noProc.result()
                        print noProc
```

```
                                except TimeoutError as e:
                                        None
                        print "//////////////////////////////////\n"

        def getNId(self):
                self.nId = self.nId+1
                return self.nId


if __name__ == "__main__":
        set_context()
        host = create_host('http://127.0.0.1:6969')

        g = host.spawn('group', Group)
        g.init_start()

        serve_forever()
```

Peer:

```
from pyactor.context import interval, later, set_context, create_host, shutdown
from random import choice
from random import randint
from Queue import PriorityQueue
from pyactor.exceptions import TimeoutError
from group import *


class Peer(object):
        _tell = ['init_start', 'afegir_group', 'join', 'stop_interval', 'leave', 'multicast', 'receive', 'process_msg',
'afegir_sequencer', 'setTimeStamp', 'voteNewSeq', 'stopPeer', 'multicast2']
        _ask = ['getId', 'getMessage', 'getFalten', 'getTimeStamp', 'getVote', 'getNextMessage', 'getUrl']
        _ref = ['afegir_group', 'afegir_host', 'leave', 'afegir_sequencer']

        def __init__(self):
                self.data = []
                self.falten = PriorityQueue()
                self.nextMessage = 0
                self.timeStamp = -1
                self.nId = 0
                self.url = None
                self.seqId = 0

        def init_start(self, url):
                self.url = url
                self.nId = self.group.getNId()
                self.group.afegirPeer(self.nId, self.url)
                self.interval1 = interval(self.host, 3, self.proxy, 'join')

        def stop_interval(self):
```

```
        self.interval1.set()

def leave(self):
        self.stop_interval()
        self.group.esborrar(self.url)
        self.group.esborrarId(self.nId)
        self.host.stop_actor(self.id)

def join(self):
        self.group.announce(self.url)

def afegir_group(self, group):
        self.group = group

def afegir_sequencer(self, sequencer, seqId):
        seq = host.lookup_url(sequencer, Peer, future=True)
        seq = seq.result()
        self.sequencer = seq
        self.seqId = seqId

def getId(self):
        return self.id

def getMessage(self):
        return self.data

def getFalten(self):
        return self.falten.qsize()

def multicast2(self, message):
        #print self.nId
        #print self.seqId
        votacio = False
        if(self.nId!=self.seqId):
                #print "NO"
                try:
                        order = self.sequencer.getTimeStamp()

                except TimeoutError as e:
                        if(self.group.getVotant() == False):
                                self.group.setVotant(True)
                                print('----------------------')
                                print('Votant nou sequencer')
                                print('----------------------')
                                self.voteNewSeq()
                                sleep(3)
                                votacio = True

        else:
                #print "SI"
                order = self.getTimeStamp()

        sleep(15)
        if(votacio == False):
```

```
                for peerUrl in self.group.getMembers():
                        try:
                                peer = self.host.lookup_url(peerUrl, Peer, future=True)
                                peer = peer.result()
                                #print message
                                #print order
                                peer.receive(order, message)
                        except TimeoutError as e:
                                None
        else:
                None

def multicast(self, message):
        #print self.nId
        #print self.seqId
        votacio = False
        if(self.nId!=self.seqId):
                #print "NO"
                try:
                        order = self.sequencer.getTimeStamp()

                except TimeoutError as e:
                        if(self.group.getVotant() == False):
                                self.group.setVotant(True)
                                print('----------------------')
                                print('Votant nou sequencer')
                                print('----------------------')
                                self.voteNewSeq()
                                sleep(3)
                                votacio = True

        else:
                #print "SI"
                order = self.getTimeStamp()

        if(votacio == False):
                for peerUrl in self.group.getMembers():
                        try:
                                peer = self.host.lookup_url(peerUrl, Peer, future=True)
                                peer = peer.result()
                                #print message
                                #print order
                                peer.receive(order, message)
                        except TimeoutError as e:
                                None
        else:
                None

def voteNewSeq(self):
        peerId = self.nId
        nouSeq = self
        ordre = self.nextMessage
        guanyo = False
        peers = self.group.getPeerIds().items()
```

```
		peers = sorted(peers, reverse=True)

		for peerId, peerUrl in peers:
			try:
				#print peerId
				if(peerId != self.nId):
					peer = self.host.lookup_url(peerUrl, Peer)
					peerIdAux = peer.getVote()
					ordreAux = peer.getNextMessage()
					nouSeqUrl = peerUrl
					nouSeqId = peerId
					break
				else:
					guanyo = True
					break
			except:
				None

		if(guanyo):
			ordre = ordre-1
			nouSeq.setTimeStamp(ordre)
			nouSeqUrl=self.url
			nouSeqId=self.nId
		else:
			ordreAux = ordreAux-1
			peer.setTimeStamp(ordreAux)
		for peerUrl in self.group.getMembers():
			try:
				peer = self.host.lookup_url(peerUrl, Peer, future=True)
				peer = peer.result()
				peer.afegir_sequencer(nouSeqUrl, nouSeqId)
			except TimeoutError as e:
				None
		self.group.setVotant(False)

	def receive(self, order, message):
		if(self.nextMessage == order):
			self.process_msg(message)
		else:
			self.falten.put((order, message))

	def process_msg(self, message):
		self.data.append(message)
		self.nextMessage+=1
		if not self.falten.empty():
			msg = self.falten.get()
			if (msg[0] == self.nextMessage):
				self.process_msg(msg[1])
			else:
				self.falten.put(msg)

	def getTimeStamp(self):
		self.timeStamp+=1
		return self.timeStamp
```

```
        def setTimeStamp(self, newTS):
                self.timeStamp = newTS

        def getVote(self):
                return self.nId

        def getNextMessage(self):
                return self.nextMessage

        def stopPeer(self):
                self.host.stop_actor(self.id)

        def getUrl(self):
                return self.url


if __name__ == "__main__":
        set_context()
        url = 'http://127.0.0.1:6970/'
        host = create_host(url)

        numPeers = raw_input("Quants peers vols al sistema? (Sense contar al sequencer)\n")

        peers = []

        g = host.lookup_url('http://127.0.0.1:6969/group', Group, future=True)
        g = g.result()

        seqId = 0
        seqUrl = url+'seq'
        seq = host.spawn('seq', Peer)
        peers.append(seq)
        peers[0].afegir_group(g)
        peers[0].afegir_sequencer(seqUrl, seqId)
        peers[0].init_start(seqUrl)
        #print peers[0].getUrl()

        for i in range (1, int(numPeers)+1):
                nom = 'peer'+str(i)
                peerUrl = url+nom
                peers.append(host.spawn(nom, Peer))
                peers[i].afegir_group(g)
                peers[i].afegir_sequencer(seqUrl, seqId)
                peers[i].init_start(peerUrl)
                #print peerUrl

        for n in range (1, 1000):
                sleep(3)
                g.infopeer()
                x = raw_input("que vols fer? (1-enviar missatge,  2-enviar missatge amd delay,  3-fer sortir peer del
sistema,  4-aturar peer directament,  5-info dels peers,  6-sortir\n")

                if(int(x) == 1):
```

```
                nomPeer = raw_input("diguem quin peer vols que envii el missatge (0-sequencer inicial, 1-peer1, 2-peer2, ...\n")
                missatge = raw_input("diguem quin missatge vols enviar\n")
                peers[int(nomPeer)].multicast(missatge)

        if(int(x) == 2):
                nomPeer = raw_input("diguem quin peer vols que envii el missatge (0-sequencer inicial, 1-peer1, 2-peer2, ...\n")
                missatge = raw_input("diguem quin missatge vols enviar\n")
                peers[int(nomPeer)].multicast2(missatge)

        if(int(x) == 3):
                sPeer = raw_input("diguem quin peer vols fer sortir del sistema\n")
                peers[int(sPeer)].leave()

        if(int(x) == 4):
                esbPeer = raw_input("diguem quin peer vols esborrar\n")
                peers[int(esbPeer)].stopPeer()

        if(int(x) == 5):
                g.infopeer()

        if(int(x) == 6):
                shutdown()
                break

        #g.infopeer()

shutdown()
```