Algoritmos e Sistemas Distribuídos 2021- Project Phase 1

Rodrigo Gomes 52596
Diogo Barreiros 52412
Gonçalo Severino 52999

Abstract

Este relatório apresenta um projeto que teve como objetivo o desenvolvimento de um protocolo utilizado para o armazenamento e compartilhamento de recursos, que utiliza uma hash table distribuída para tal efeito. Após o desenvolvimento do sistema, este será testado para analisar o seu desempenho relativamente a um conjunto de testes, variando a sua complexidade. Este projeto é uma demonstração da implementação de um sistema distribuído, comunicações entre diferentes processos e comunicações entre diferentes *layers* dentro de um mesmo processo.

Introdução

Um dos principais objetivos do projeto é construir um sistema, ponto a ponto, de compartilhamento e armazenamento de recursos. Este projeto também tem em vista o desenvolvimento de protocolos e algoritmos para uma hash table distribuída ponto a ponto.

O sistema será testado numa máquina local e também com um cluster computacional de modo que sejam observadas as suas diferenças de desempenho e custos.

Assim este relatório irá ser constituído por quatro secções, em complemento desta introdução. A primeira é destinada à apresentação dos diferentes pseudo-codes explicativos dos métodos implementados, juntamente com a descrição respetiva. Na secção seguinte serão apresentados os resultados relativos à testagem do projeto desenvolvido, assim como a sua análise. Por fim, este relatório terminará com uma breve conclusão.

Implementação

Foi inicialmente feito um estudo do exemplo, disponibilizado nas aulas práticas, de uma implementação usando o Babel. Foi avaliada a forma usada para implementação da interface referente aquele sistema distribuído e consequentemente foi pensado como poderíamos fazer implementação do nosso projeto.

Para implementação do *Storage Protocol* foi tida em conta a interface disponibilizada pelo professor, e as classes já definidas, sendo que decidimos utilizar dois *HashMaps* referentes aos conteúdos/ficheiro gerados na aplicação, sendo que um deles com entradas do tipo *Nome do Ficheiro->Conteúdo* onde estarão guardados os documentos do nó local, e outro mapa, que percebemos ser necessário ao longo da realização do projeto, também relativo aos documentos locais mas neste caso com entradas do tipo *Id do Ficheiro–>Nome do Ficheiro*, sendo que sempre que criamos uma nova entrada num mapa fazemos o mesmo no outro e do mesmo modo quando pretendemos apagar uma entrada. Além dos métodos da interface presentes no enunciado criamos também os métodos que tratam dos *Bulk Requests*, "*Bulk*" devido aos movimentos de volumes de ficheiros necessários quando na DHT um novo nó se junta ao anel.

Sobre o protocolo de DHT foi apenas implementado o protocolo *Chord.* Todo o seu desenvolvimento teve por base o artigo disponibilizado nas aulas "Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications" e o pseudocódigo ali escrito. Usamos para a Tabela de *Fingers* um mapa com entradas do tipo *Id(BigInteger)->Host do Nó Relativo* , sendo este Host encontrado na função auxiliar *findNodeFile(BigInteger id),* que decidimos implementar, e que como referido no artigo vai fazer a computação necessária para descobrir a entrada da tabela de fingers com o id inferior mas mais próximo do id passado em argumento. Foi criado também um set de Host's, chamado *connections*, que contém os hosts para o qual o nó local tem conexões ligadas, para assim no método da verificação do predecessor (*uponCheckPredecessorTimer*) podermos

consultar se existe conexão para este, e no caso de não ser verdade assumirmos que falhou. Além destas estruturas foi necessária a criação de um set de *BigIntegers*, chamado *filesLocal,* para guardar os ids dos ficheiros guardados no nó local, e facilitar as computações relativas aos pedidos vindos do *storage protocol* de *store*, *retrieve* e também na passagem de ficheiros de um nó para outro quando um novo nó se junta à rede, sendo possível, por tudo isso, diminuir pedidos ao *storage protocol* que iriam gastar mais recursos de comunicação desnecessariamente.

Todo o processo de implementação evoluiu naturalmente, as maiores dificuldades encontradas foram no início, para adaptação aos elementos e ambiente do Babel, e posteriormente, na perceção da lógica necessária para inicialização do anel referente ao protocolo Chord. À exceção destes pontos, e tendo em conta a dificuldade em desenvolver computações para sistemas distribuídos, tudo fluiu naturalmente.

O pseudocódigo referente à implementação do *storage protocol* e do *chord protocol* encontra-se em anexo.

Trabalho Experimental

Os testes foram realizados utilizando o script "start-processes.sh" com o protocolo Chord. Foram testadas as combinações de dois valores diferentes de *payload_size* e de *request_interval*. As seguintes definições mantiveram-se iguais em todas as execuções: *content_number* = 4, *total_processes* = 4, *prepare_time* = 10, *cooldown_time* = 10 e *run_time* = 60.

O código entregue tem um erro no cálculo do retrieve latency. Estes testes foram realizados numa versão semelhante, com as devidas alterações no AutomaticApplication para o correto cálculo deste valor.

A Tabela 1- anexo 3 - apresenta as médias dos valores dos vários nós para combinações de *payload_size* igual a 100 e a 500 bytes com *request_interval* de 1000 e de 500 milissegundos.

Os valores de *Recall Rate* e de *Retrieve Latency* do teste C e de *Recall Rate* do teste D parecem ser *outliers*, isto deve-se ao facto dos testes serem apenas uma execução do programa. Para maior robustez de resultados deveria ter sido usada a média de várias execuções.

O número elevado de mensagens enviadas e recebidas deve-se a um bug na nossa implementação onde algumas mensagens estão a ser enviadas ciclicamente.

A diminuição *Recall Rate* do teste D pode ser explicada pelo aumento do "peso" da execução devido ao *payload* maior e a um maior número de *requests*. Algumas falhas em pedidos, e a consequente diminuição do *Recall Rate*, pode ser explicada pelo *AutomaticApplication* de um nó fazer pedidos de ficheiros que podem ainda não ter sido criados por outro nó.

Achamos que os valores usados nos testes não são suficientemente díspares para observar comportamentos diferentes no programa. Isto provém de algum problema na implementação onde com parâmetros altos de *content_number*, *total_processes*, *payload_size* e/ou com valores baixos de *request_interval*, a execução de alguns nós não termina, impossibilitando o cálculo de alguns valores estatísticos.

Conclusão

Com este projeto foi possível perceber a complexidade de implementação de um sistema distribuído devido à constante necessidade de se perceber o que os próximos nós/processos deverão fazer quando ocorre a propagação de uma mensagem. Percebemos também que testar este tipo de programas é bastante difícil devido ao grande número de processos que funcionam ao mesmo tempo e ao grande número de ações que realizam.

No entanto, todo o projeto se revelou muito útil pois apesar destas dificuldades, foi possível adquirir vastos conhecimentos, desde como implementar comunicações entre diferentes processos à implementação de comunicações

entre diferentes *layers* dentro de um mesmo processo, além da aprendizagem do protocolo *Chord*.

Anexos

```
STORAGE

State:
    dhtProtoId
    upProtoId
    contentsMap
    idToName

    Upon ChannelCreated({ChannelCreated, notification}) do:
        Trigger registerSharedChannel(notification.getChannelId())

    Upon StoreRequest({StoreRequest, request}) do:
        Trigger sendRequest(request, dhtProtoId)

    Upon StoreLocalRequest({StoreLocalRequest, name, content}) do:
        contentsMap <- contentsMap U {(name, content)}
        id <- GenerateHash(name)
        idToName <- idToName U {(id, name)}

    Upon LookupLocalRequest({LookupLocalRequest, name, host, content}, sourceProto) do:
        content <- contentsMap[name]
        if(content == {}) then
            Trigger sendReply(LookupLocalReply, {name, host, null}, sourceProto)
        else then
            Trigger sendReply(LookupLocalReply, {name, host, content}, sourceProto)

    Upon LookupBulkLocalRequest({LookupBulkLocalRequest, files, host}, sourceProto) do:
        names // local set with the name of the files
        contents // local set with the contents of the requests
        foreach (id) E files do:
            if(id != ⊥) then
                names <- names U {(idToName[id])}
                contents <- contents U {(contentsMap[idToName[id]])}
        Trigger sendReply(LookupBulkLocalReply, {host, names, contents}, sourceProto)

    Upon StoreBulkLocalRequest({StoreBulkLocalRequest, names, contents, host}, sourceProto) do:
        foreach name E names do:
            foreach content E contents do:
                contentsMap <- contentsMap U {(name, content)}
                idToName <- idToName U {(GenerateHash(name), name)}

        Trigger sendReply(StoreBulkLocalReply, {host, name}, sourceProto)

    Upon RemoveBulkRequest({RemoveBulkRequest, ids}) do:
        foreach (id) E ids do:
            if (idToName[id] != ⊥) then
                contentsMap <- contentsMap \ {idToName[id]}
                idToName <- idToName \ {id}

    Upon RetrieveRequest({RetrieveRequest, name, UID}) do:
        content // local byte array
        content <- contentsMap[name]
        if(content != ⊥) then
            Trigger sendReply(RetrieveOKReply, {name, UID, content}, upProtoId)
        else then
            Trigger sendRequest(LookupRequest, {GenerateHash(name), name}, dhtProtoId)

    Upon LookupOkReply({LookupOKReply, name, UID, content}) do:
        Trigger sendReply(RetrieveOKReply, {name, UID, content}, upProtoId)

    Upon LookupFailedReply({LookupFailedReply, name, UID}) do:
        Trigger sendReply(RetrieveFailedReply, {name, UID}, upProtoId)

    Upon StoreOkReply({StoreOKReply, reply}) do:
        Trigger sendReply(reply, upProtoId)

    UponLookupResponseMsg({LookupResponseMsg, content, name, Uid}) do:
        if(content == {}) then
            Trigger sendReply({LookupFailedReply, name, Uid}, storageProtoId);
        else then
            Trigger sendReply({LookupOKReply(name, Uid, content)}, storageProtoId);

    Upon CheckPredecessorTimer() do:
        if(connections /E preHost) then
            preHost <- ⊥;
            preId <- ⊥;
```

```
77
78      Procedure between(left, middle, right) do:
79          if(left < right) then
80              return (middle > left && middle < right)
81          else then
82              return (middle > left || middle < right)
83
84      Procedure findNodeFile(id){
85          host <- sucHost
86          maxId <- sucId
87
88          foreach (key, value) E fingerTable do:
89              if (Call between(maxId, key, id)) then
90                  maxId <- key
91                  host <- value
92
93          return host
94
95      Upon OutConnectionUp({OutConnectionUp, node}) do:
96          connections <- connections U {node}
97
98      Upon OutConnectionDown({OutConnectionDown, node}) do:
99          connections <- connections \ {node}
```

Anexo 1

```
1   ChordProtocol
2
3   State:
4       fingerTable //HashMap com entradas do tipo Key(BigInteger)-Value(Host)
5       connections //set de peer a quem o nó local está conectado
6       filesLocal //lista de BigInteger referentes aos ids dos conteúdos que
7                  estão guardados pelos nós locais
8       sucHost
9       sucId
10      preHost
11      preId
12      channelId
13      selfId
14      selfHost
15      storageProtoId
16      next
17
18      Upon Init(props) do:
19          selfHost <- self
20          selfId <- Call GenerateHash(self)
21          storageProtoId <- storageProtoId
22          next <- 0
23          channelId <- CreateChannel
24          Connections <- { }
25          filesLocal <- { }
26          fingerTable <- { }
27          Trigger Notification(channelId)
28          preHost <- null
29          preId <- null
30          if( contact E props  ) then
31              Trigger openConnection(contactHost,channelId)
32              Trigger SendMessage (channelId,{ FindSucMsg, selfHost,selfId,seldId},contactHost  )
33          Else then
34              sucHost <- selfHost
35              sucId <- selfId
```

```
36
37      Upon FindSucMsg( { FindSucMsg ,host,hashId,targetId}) do:
38          If  ( sucId == selfId  || Call Between(selfId, targetId, sucId)) them
39              Trigger openConnection(host,channelId)
40              Trigger SendMessage (channelId,{ FindSucMsgResponse, sucHost,sucId,targetId},host  )
41          Else then
42              destinyHost <- Call findNodeFile(targetId)
43              Trigger SendMessage (channelId,{ FindSucMsg, selfHost,selfId,seldId}, destinyHost)
44
45      Upon FindSucMsgResponse( ,{ FindSucMsgResponse, host,hashId,targetId} )  do:
46          If( selfId == targetId) then
47              sucId <- hashId
48              sucHost <- host
49              Trigger sendMessage(channelId, { AskForContentMsg, selfId},sucHost)
50          else then
51              fingerTable[targetId] <- host
52
53      Upon AskForContentMsg({AskForContentMsg, hashId},from) do:
54          Files <- {} //local
55          forEach fileId E filesLocal do:
56              if(fileId < hashId  || fileId > selfId) then
57                  files U {fileId}
58          if(files != {} ) do:
59              Trigger sendRequest({LookUpBulkLocalRequest ,from,files}, storageProtoId )
60
61      Upon LookUpBulkLocalReply( {LookupBulkLocalReply, host, names, contents }) do:
62          Trigger sendMessage(channelId, { StoreBulkMsg ,names, contents },host)
63
64      Upon StoreBulkMsg({ StoreBulkMsg ,names, contents},from) do:
65          Trigger sendRequest({StoreBulkLocalRequest, from,names, contents },storageProtoId)
66
67      Upon StoreBulkLocalReply( { StoreBulkLocalReply, host,names } ) do:
68          Ids <- {} // local
69          forEach name E names do:
70              id <- GenerateHash(name) // local
71              ids U {id}
72              filesLocal U {id}
73          Trigger sendMessage( channelId, {ConfirmBulkRequest, ids}, host )
74
75      Upon ConfirmBulkStoreMsg({ConfirmBulkStoreMsg, fileIds}) do:
76          ForEach id E fields do:
77              filesLocal \ {id}
78          Trigger sendRequest( {RemoveBulkRequest, fileIds}, storageProtoId  )
79
80      Upon StabilizeTimer(StabilizeTimer timer) do:
81          Trigger openConnection(sucHost, channelId)
82          Trigger sendMessage(channelId, { StabilizeMsg ,selfHost, selfId }, sucHost  )
83
84      Upon StabilizeMsg( {StabilizeMsg, host,id }, from ) do:
85          Trigger openConnection(from, channelId)
86          If( preId == _I_ || Call Between(preId, id,selfId)) then
87              preId <- id
88              preHost <- host
89          Trigger sendMessage(channelId, { StabilizeMsgResponse ,preHost, preId },from )
90
91      Upon StabilizeMsgResponse({StabilizeMsgResponse,host,id}) do:
92          If(Call Between(selfId, id, sucId)) then
93              sucId <- id
94              sucHost <-host
95
96      Upon FixFingersTimer( { FixFingersTimer, timer}) do:
97          m <- 2 //local
98          next <- next +1
99          if (next > m) then
100             next <- 1
101         targetKey <- selfId + 2^(next+1) //local
102         contactHost <- Call findNodeFile(tarketKey)
103         if(contactHost == sucHost) then
104             fingerTable[targetKey] <- sucHost
105         else then
106             Trigger openConnection(contactHost, channelId)
107             Trigger sendMessage(channeld, {FindSucMsg(selfHost, selfId, targetKey), contactHost})
```

```
108
109     Upon StoreRequest({StoreRequest,uuid, name, content} ) do:
110         fileId <- GenerateHash(name) //local
111         if(predId == _I_ || Call Between(preId,fileId,selfId)) then
112             Trigger sendRequest( {StoreLocalRequest, name, content }, storageProtoId)
113             Trigger sendReply({StoreOkReply, name, uuid},storageProtoId)
114             filesLocal U {fileId}
115         else if( sucHost != _I_) then
116             targetHost <- Call findNode(fileId) //local
117             Trigger sendMessage({StoreMsg,selfHost,fildeId,name,content, uuid}, targetHost)
118
119     Upon StoreMsg({StoreMsg,host,fileId,name,content,uuid}) do:
120         If( Call Between(preId, fileId,selfId)) then
121             Trigger sendRequest({StoreLocalRequest, name, content}, storageProtoId)
122             Trigger sendMessage(channelId, {StoreSuccessMsg, name, uuid},host)
123             filesLocal U {fileId}
124         else then
125             targetHost <- Call findNode(fileId) //local
126             Trigger sendMessage(channelId, {StoreMsg,host,fileId,name,content,uuid}, targetHost)
127
128     Upon StoreSuccessMsg({StoreSuccessMsg, name, uuid},host)) do:
129         Trigger sendReply({StoreOkReply,name,uuid},storageProtoId)
130
131     Upon LookupRequest({LookUpRequest, id, name,uuid}) do:
132         targetHost <- Call findNode(fileId) //local
133         Trigger sendMessage(channelId, {LookupMsg, selfHost,fileId,name, uuid}, targetHost)
134
135     Upon LookUpMsg({LookupMsg, selfHost, fileId,name, uuid}) do:
136         If (predId != _I_ && Call Between(predId,fileId,selfId)) then
137             If (fileId E filesLocal) then
138                 Trigger SendRequest({LookUpLocalRequest, name, host},storageProtoId)
139         Else if (sucHost != _I_) then
140             targetHost <- Call findNode(fileId) //local
141             Trigger sendMessage(channelId, {LookupMsg, selfHost, fileId,name, uuid}, targetHost)
142
143     Upon LookupLocalReply({LookupLocalReply , name, host, content, uid}) do:
144         Trigger sendMessage(channelId,{LookUpResponseMsg, name,uid,content }, host)
145
146     UponLookupResponseMsg({LookupResponseMsg, content, name, Uid}) do:
147         if(content == {}) then
148             Trigger sendReply({LookupFailedReply, name, Uid}, storageProtoId);
149         else then
150             Trigger sendReply({LookupOKReply(name, Uid, content)}, storageProtoId);
151
152     Upon CheckPredecessorTimer() do:
153         if(connections /E preHost) then
154             preHost <- ⊥;
155             preId <- ⊥;
156
157     Procedure between(left, middle, right) do:
158         if(left < right) then
159             return (middle > left && middle < right)
160         else then
161             return (middle > left || middle < right)
162
163     Procedure findNodeFile(id){
164         host <- sucHost
165         maxId <- sucId
166
167         foreach (key, value) E fingerTable do:
168             if (Call between(maxId, key, id)) then
169                 maxId <- key
170                 host <- value
171
172         return host
173
174     Upon OutConnectionUp({OutConnectionUp, node}) do:
175         connections <- connections U {node}
176
177     Upon OutConnectionDown({OutConnectionDown, node}) do:
178         connections <- connections \ {node}
```

Anexo 2

Test	Payload Size	Request Interval	Recall Rate	Store Latency (ms)	Retrieve Latency (ms)	Msg send	Bytes transmitted	Msg receive	Bytes received
A	100	1000	0.756	1386.75	27.76	1318	201144	1318	201144
B	100	500	0.771	216.5	26	1667	226674	1667	226674
C	500	1000	0.983	147.5	7.5	956	243212	956	243212
D	500	500	0.591	182.5	38.6	2130	325635	2130	325635

Anexo 3 - Tabela 1 - Resultados dos testes com o protocolo Chord