Computação Distribuída e Colaborativa na Periferia

Rúben Barreiro, Pedro Sanches, Filipe Cerqueira, and Hervé Paulino

NOVA Laboratory for Computer Science and Informatics
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa, 2829-516 Caparica, Portugal
{r.barreiro,p.sanches,fa.cerqueira}@campus.fct.unl.pt
herve.paulino@fct.unl.pt

Abstract. A Computação *Edge* propõe o deslocamento de (algumas) tarefas de computação dos serviços *Cloud* centralizados para perto dos utilizadores, aproveitando o poder de computação e armazenamento dos computadores e dispositivos móveis atuais. Consequentemente, a computação vai ser tipicamente efetuada de forma colaborativa por vários nós que, dependendo do seu fluxo atual, podendo consumir computação de outros nós ou produzir computação para os mesmos. No entanto, os nós podem entrar ou sair da rede, forçando o sistema a ser resilente a situações de *churn*. Este paper apresenta um sistema para computação distribuída colaborativamente numa rede composta por diferentes nós. O sistema é construído sobre uma fila de tarefas, replicada, distribuída e eventualmente consistente, embora com pouca coordenação. A nossa proposta é avaliada sobre o contexto de computação móvel, demonstrando que a nossa solução é linearmente escalável e suporta situações de *churn* de forma eficiente.

1 Introdução

Observando o atual quotidiano social, podemos afirmar que a tecnologia está cada vez mais presente na vida das pessoas. A Internet sofreu uma evolução enorme nos últimos anos e o seu número de utilizadores, tal como a quantidade de dados processados continua a crescer. E muitas das aplicações atuais que fazem uso da mesma, precisam de grandes transferências entre os dispositivos e os centros de dados. De acordo com a *Cisco*, em 2021, o número total de *smartphones* (incluindo *phablets*) representará 50% do total de dispositivos e ligações móveis à escala global (6.200 milhões), dos 3.600 milhões em 2016.

Com o eminente crescimento deste paradigma e para certos tipos de dispositivos, começa a ser importante fazer o processamento dos dados localmente, numa noção de periferia onde estarão outros dispositivos semelhantes, em vez de fazer o processamento baseado em arquiteturas e sistemas que funcionem com base em centros de dados comuns, onde os dados podem ser armazenados e processados, como os servidores de Sistemas Distribuídos e Sistemas *Cloud*

atuais. Estes centros de dados muitas vezes estão afastados dos utilizadores, introduzindo latência, tal como também, são constantemente sobrecarregados com operações e dados, congestionando toda a rede.

Esta nova abordagem de trazer o processamento dos dados para perto dos dispositivos dos utilizadores, pode ser endereçada pela Computação *Edge*. Como os dispositivos móveis atuais têm algumas limitações de energia e podem precisar de processar maior quantidade de dados, beneficiariam de realizar *crowdsourcing* dos recursos dos dispositivos mais próximos. Estes dispositivos irão produzir e consumir dados de forma colaborativa, estando, na sua maioria, perto uns dos outros geograficamente.

Por exemplo, com esta tecnologia, numa cidade com uma grande densidade de dispositivos, se a rede de Internet falhasse, poderia permitir aos dispositivos comunicarem entre eles, por um breve período de tempo, ao remover sobrecarga dos pontos de acesso, fazendo toda a comunicação localmente. Por exemplo, num jogo de futebol, se uma pessoa gravasse um golo com o seu *smartphone*, poderia partilhá-lo com outras pessoas próximas dela, mesmo que não existissem redes Internet disponíveis, naquele momento.

O problema desta abordagem, está na volatilidade da rede, dada a possível mobilidade dos dispositivos. Os dispositivos podem entrar e sair da rede local a qualquer momento e não existem pontos centrais de sincronização ou mesmo mecanismos que tenham sempre conhecimento da composição da rede e dos dispositivos que a compõem, em qualquer momento.

Nesse sentido, atualmente está a ser desenvolvido um projeto de investigação para um algoritmo de gestão de uma fila distribuída de tarefas. Este algoritmo é baseado em replicação com consistência eventual e não precisa de qualquer conhecimento da rede e sua composição. O objetivo deste projeto de investigação é a implementação deste algoritmo para dispositivos móveis que usem sistemas operativos *Android*.

Os objetivos deste trabalho são:

i) Uma proposta e implementação de um protótipo para uma *framework* para computação distribuída em redes compostas por dispositivos móveis que usam sistemas operativos *Android*, que permitam armazenar e distribuir, com pouca coordenação, tarefas de computação, para todos os nós na rede, sem a noção de um componente centralizado(Section 2). Esta *framework* usa uma estrutura de dados, mais especificamente, uma fila que irá ser replicada e distribuída por todos os nós, que irá armazenar tarefas de computação, que poderão ser executadas por qualquer nó na rede.

2 DiCE

DiCE é um sistema completamente distribuído para dispositivos que realizarão computação na periferia, colaborando entre si na execução de múltiplas tarefas.

No entanto, múltiplos cenários são possíveis, aqui estão a ser endereçados particularmente ambientes de redes de dispositivos móveis (*smartphones*, *tablets* ou *phablets*), como grupos de *WiFi Direct*, ou conjuntos de dispositivos conectados

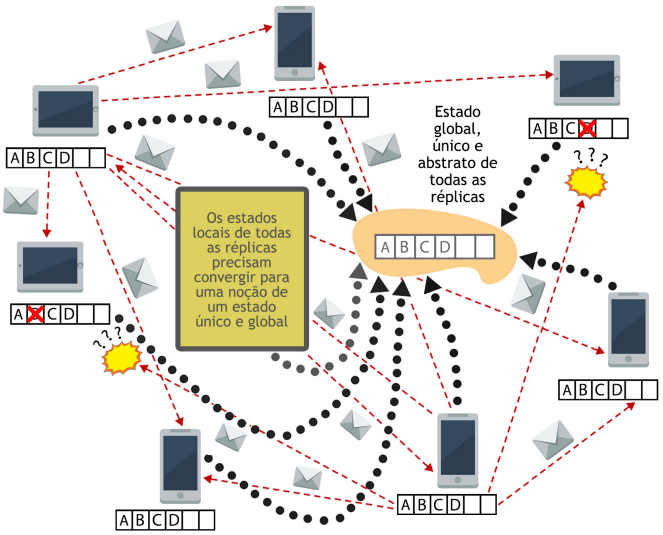

Fig. 1: Uma rede *DiCE* colaborando para realizar a execução de 4 tarefas: A, B, C e D

a pontos de accesso *WiFi* e em redes locais (*LANs*). Este sistema, é portanto, direcionado para redes dinâmicas, expostas a situações de *churn*, que podem incluir dezenas ou centenas de de dispositivos, sem restrições de mobilidade.

Dada a volatilidade natural dos cenários descritos anteriormente, armazenar todas as operações efetuadas pelos diferentes nós, exige muita coordenação por parte dos mesmos. Para evitar a sobrecarga imposta por esta eventual coordenação, os dispositivos comunicam entre si através de mensagens *Broadcast* (atráves de intermediário com ou sem fios), mas sem saber o estado nem a composição da rede. O tráfego imposto pelas emissões de mensagens por *Broadcast* precisa alcançar todos os dispositivos na rede, contudo os serviços de comunicação por *Broadcast* não são fiáveis e podem ocorrer algumas perdas de mensagens.

Para limitar ainda mais a necessidade de coordenação, cada dispositivo possui uma réplica de uma fila replicada eventualmente consistente que armazena as tarefas atualmente disponíveis no sistema: as que estão à espera de serem executadas, as que estão em execução pelo respetivo dispositivo, as que estão a ser executadas por um outro dispositivo, as que estão a aguardar o retorno do resultado de uma execução por um outro dispositivo e as que já tiveram a sua execução terminada por algum dispositivo.

A Figura 1 descreve uma visão geral do sistema, onde todos os dispositivos mantêm uma réplica do estado atual da fila, e efetuam comunicação por *Broadcast*, podendo ocorrer, por vezes, perdas de mensagens, que poderam originar

situações que façam com que os diferentes estados replicados possam divergir, e com isso, a necessidade para definir mecanismos para que se garanta que todos os estados da estrutura replicados, convirjam para um estado único e singular.

É assumido que cada dispositivo possui um identificador globalmente exclusivo e único, e todos os dispositivos têm os seus relógios sincronizados (com uma diferença insignificante), que é um pressuposto razoável tanto em redes *LAN* como em redes de dispositivos móveis. Por exemplo, no último caso, os dispositivos geralmente sincronizam os seus relógios com o tempo fornecido pela operadora provedora da rede. Neste sistema, também é considerado o modelo de falhas clássico *crash-stop*, no qual os dispositivos podem falhar por crashes, mas não têm comportamentos maliciosos. No que se refere às tarefas, assume-se que estas são idempotentes, no sentido de que a sua execução devolve sempre o mesmo resultado, independentemente do dispositivo que a realizou e do número de vezes que a operação é acionada. As tarefas também têm de possuir a propriedade de serem comutativas, ou seja, um dado conjunto de operações envolvendo as mesmas, tem de produzir sempre o mesmo resultado, independentemente da ordem pelas quais são executadas. Além disso, as tarefas devem implementar o seguinte conjunto de funções:

- *UUID getElementID()* - Obtenção do identificador de uma tarefa;
- *boolean isExecutionConcluded()* - Verificação de, se a tarefa já foi executada ou não;
- *Map<Address, Integer> getNumObjectsByUser()* - Obtenção dos identificadores dos dispositivos que possuem dados/objetos associados a esta tarefa e a quantidade agregada dos mesmos, a cada dispositivo;

Nesta secção, é apresentada a pilha de software e todos os mecanismos de *crowdsourcing*, disponíveis e executados por cada dispositivo localmente. A Figura 2 ilustra essa mesma pilha de *software* oferecida pelo sistema DiCE e as suas respetivas camadas.

De referir que o sistema DiCE faz uso de quatro estruturas de dado auxiliares, que são as seguintes:

- *Fila/Lista de Tarefas Prontas a Executar* - Representa a fila distribuída e replicada por cada dispositivo em si e é a estrutura de dados príncipal do sistema. Contém todas as tarefas que foram inseridas no sistema por um dos dispositivos e que estão à espera de serem executadas por um dos dispositivos presentes na rede;
- *Mapa de Tarefas Localmente Conhecidas* - Contém todas as tarefas localmente conhecidas, das quais se possui dados e que ainda não tiveram a sua executada terminada por nenhum dispositivo na periferia;
- *Mapa de Tarefas em Execução* - Contém todas as tarefas das quais se tem conhecimento de estarem a serem executadas por outros dispositivos na periferia que não o próprio, recebendo mensagens periódicas sobre a sua execução (mensagens *heartbeat*);
- *Mapa de Tarefas já Executadas* - Contém todas as tarefas sobre as quais se tem conhecimento localmente de já terem a sua execução terminada pelo próprio dispositivo ou por algum dos presentes na periferia;

Fig. 2: Arquitetura do sistema DiCE

2.1 *Framework* API

O DiCE é um sistema simétrico, onde cada dispositivo móvel executa a pilha de software (ilustrada na Figura **??**). É importante referir que esta *framework* encontra-se a ser desenvolvida no contexto do projeto *Hyrax* [**?**], utilizando diferentes componentes/sub-sistemas que foram ou estão a ser desenvolvidos por outras equipas, no mesmo contexto.

DiCE *API*. Esta camada é a interface da *framework* oferecida ao cliente e que fornece as principais operações que o mesmo pode efetuar sobre a estrutura de dados local. Aqui podem ser executadas as operações de adicionar novas tarefas ao sistema, remover tarefas presentes no sistema para execução local, cancelar uma execução corrente de uma tarefa, tal como remover tarefas do sistema que se pretende que não sejam de todo processadas por algum dispositivo. As operações mencionadas anteriormente podem ser descritas pelas seguintes assinaturas:

- submit(Element element) – Submete uma nova tarefa no sistema;
- removeByID(UUID elementID) – Remove uma tarefa anteriormente submetida e presente no sistema, dado o seu identificador;

De referir que, por cada execução de uma destas operações, o dispositivo que a efetuou, notifica os restantes dispositivos na periferia, por emissão *Broadcast*, acerca da realização das mesmas, de forma a que os outros dispositivos tenham conhecimento das mesmas e possa realizar os mecanismos adequados para cada tipo de situação, para que todo o sistema possa convergir para o mesmo estado. O módulo representado nesta camada, também segue a interface comum das filas com as operações de *enqueue*, *dequeue*, *remove*, *peek*, *isFull*, *isEmpty*, e outras variantes.

Gestor de Tarefas à Espera de serem Executadas. A camada de *Gestor de Tarefas à Espera de serem Executadas* é responsável pela gestão das tarefas que estão na *Fila de Tarefas prontas a Executar* e que ainda não foram executadas por nenhum dispositivo.

Gestor de Tarefas Desconhecidas. A camada de *Gestor de Tarefas Desconhecidas* é responsável pela gestão das tarefas sobre as quais são recebidas mensagens e não se tem conhecimento e informação acerca das mesmas localmente, seja por perda de mensagens ou por alguma eventual falha do sistema. É executada ao nível do *Mapa de Tarefas Localmente Conhecidas*. Esta camada trata estas situações, enviando uma mensagem por *Broadcast* (mensagem *DOYOUKNOWTHISELEMENT*), solicitando aos outros dispositivos na periferia, o envio dos dados sobre a mesma. Os outros dispositivos, ao receberem esta mensagem e caso tenham conhecimento acerca da tarefa em questão, têm dois mecanismos para solucionarem este problema:

- Caso a tarefa ainda não tenha sido executada por nenhum dispositivo, envia os dados sobre a mesma (mensagem *TAKEMYELEMENT*);
- Caso a tarefa já tenha sido executada por algum dispositivo, é enviada uma mensagem notificando todos os dispositivos que essa mesma tarefa já foi executada por algum dispositivo em algum momento (mensagem *ALREADYFINISHEDELEMENT*);

De referir que, os dispositivos ao receberem esta mensagem e antes de prosseguirem para o envio da mensagem de resposta, consoante as situações mencionadas anteriormente, procedem a um algoritmo para determinar se eventualmente irão ser os próprios a oferecer a resposta ou um outro dos dispositivos na periferia. Este algoritmo garante que a rede local não é sobrecarregada com mensagens e tráfego desnecessários, ou seja, que apenas um dos dispositivos oferece a resposta à mensagem *DOYOUKNOWTHISELEMENT*. Este algoritmo será explicado posteriormente no algoritmo 2 da secção 2.2 (Section 2).

Sorteio e Desempate de Dequeuing/Execução. A camada de *Sorteio e Desempate de Dequeuing/Execução* é responsável pelo sorteio das candidaturas apresentadas pelos vários dispositivos para a remoção de uma tarefa para execução local, durante a janela temporal fornecida para o mesmo. Este módulo também é responsável pelo desempate entre dois dispositivos que pretendam executar a mesma tarefa num determinado momento, isto é, numa situação em que um dos dispositivos já se encontra a executar a tarefa e o outro apresentou a candidatura fora da janela temporal de candidaturas para a remoção e execução da mesma. Para determinar o dispositivo vencedor deste sorteio e/ou desempate, foi estabelecida uma relação de ordem, que é sempre executada nas situações mencionadas anteriormente e que é a seguinte:

- 1) Qual o dispositivo, de entre os candidatos, que possui mais dados ou ou objetos da tarefa que está a ser disputada para remoção e execução local?
- 2) Estado ou nível de energia da bateria dos respetivos dispositivos candidatos (ordem decrescente);

- 3) Estampilha temporal das candidaturas efetuadas pelos diversos dispositivos, para remoção e execução local (ordem crescente, da mais antiga para a mais recente);
- 4) Classificação dos respetivos dispositivos candidatos (ordem crescente);
- 5) É algum dos dispositivos, de entre os candidatos, o responsável pela introdução da tarefa no sistema?
- 6) Indentificador dos dispositivos candidatos (ordem alfanúmerica crescente);

O vencedor de cada sorteio será o dispositivo que ficará sempre responsável pela execução da tarefa a ser disputada nesse mesmo sorteio. Caso a tarefa não esteja a ser executada por nenhum nó, o nó que ficará responsável, enviará por *Broadcast* uma mensagem *DEQUEUE* aos restantes dispositivos. Este processo é explicado em detalhe no algoritmo 1 da secção 2.2 (Section 1)

Gestor de Tarefas em Execução. A camada do *Gestor de Tarefas em Execução* é a camada responsável pela gestão de todas as tarefas em execução e é executada ao nível do *Mapa de Tarefas em Execução*. Este gestão tem duas facetas: a gestão das execuções feitas por outros dispositivos e a gestão das execuções feitas pelos próprios dispositivos; Durante execução local de uma tarefa por um determinado dispositivo, são enviadas mensagens periódicas por *Broadcast* (denominadas de *heartbeats*, ou mensagens *RUNNING*), relatando o estado atual da execução aos outros dispositivos. Isto possibilita que os outros dispositivos detetem falhas (*crashes*) de dispositivos ou erros de execução por parte dos mesmos. Ou seja, caso sejam deixadas de ser recebidas estas mensagens periódicas durante a execução de uma tarefa por um determinado dispositivo, poderá ser deduzido que o mesmo falhou (*crashou*) ou que abandonou a periferia local da rede. Os dispositivos na periferia ao receberem estas mensagens periódicas, atualizam a informação relativamente às tarefas que têm conhecimento de estarem a serem executadas naquele momento por outros dispositivos. No momento em que a execução é terminada, o dispositivo responsável pela mesma envia uma mensagem por *Broadcast* (mensagem *DONE*), notificando os mesmos que esta terminou. Nesta situação, tanto o dispositivo que terminou a execução da tarefa em questão, como os dispositivos que foram notificados acerca do fim da execução da mesma, têm que adicionar a tarefa ao *Mapa de Tarefas já Executadas* e no caso dos dispositivos que recebem a mensagem *DONE*, removê-la também do *Mapa de Tarefas em Execução*.

Gestor de Comunicação. Como o nome indica, esta camada diz respeito à comunicação feita entre os vários dispositivos. Esta camada é responsável pela recepção de todas as mensagens recebidas e para cada tipo de mensagem recebida, qual o procedimento a ser tomado. Esta camada também é responsável por todas as mensagens enviadas em *Broadcast* para a rede. Nesta camada, são oferecidas um conjunto de operações assíncronas de manipulação sobre a fila:

- *enqueue(task, timeStamp)* – A tarefa *task* é inserida na fila por um determinado dispositivo no momento *timeStamp*, que é responsável por difundir os dados sobre a mesma por mensagem *Broadcast* para os restantes dispositivos na periferia;

– *dequeue(taskID, timeStamp, info)* – O dispositivo que realiza esta operação pretende remover a tarefa com o identificador *taskID* para execução local, no momento *timeStamp*. O parâmetro *info* inclui a informação sobre o dispositivo que realiza a operação, a ser usada pelo algoritmo de sorteio das candidaturas apresentadas pelos vários dispositivos para a remoção da mesma tarefa para execução local, durante a janela temporal fornecida para o mesmo;
– *removeByID(taskID)* – A tarefa *task* com o identificador *taskID* foi removida e o dispositivo responsável por esta operação envia uma mensagem *Broadcast* para todos os dispositivos na rede, informando-os acerca do mesmo;
– *info(infoType, taskID, timeStamp)* – Um dispositivo envia uma mensagem do tipo *infoType* sobre a tarefa com o identificador *taskID*. São suportados os seguintes tipos de mensagens *info*: *RUNNING*, *FAILED*, *CANCELED* e *DONE*.

2.2 Algoritmos Usados

3 Resultados Experimentais

Realizaram-se alguns testes, utilizando um simulador em Java. Foram feitos testes em ambientes com 2, 4, 8, 16 e 32 nós/dispositivos. Em alguns casos, foram detetadas algumas inconsistências no algoritmo de *Sorteio e Desempate de Dequeuing/Execução*. Ainda assim, foi possível obter alguns resultados experimentais. A figura 3 demonstra um gráfico em que um dos mesmos é representado.

Fig. 3: 4 dispositivos com 128 tarefas no sistema para executar, em que apenas um dos dispositivos realizou a operação de *ENQUEUE* das 128 tarefas, sem perdas de mensagens

Algorithm 1 Remoção de uma Tarefa

```
1: locallyKnownTasksMap:                          ▷ Mapa de Tarefas Localmente Conhecidas
2: waitToRunTasksQueue:                           ▷ Fila/Lista de Tarefas Prontas a Executar
3: dequeueRanking:                                ▷ Sorteio e Desempate de Dequeuing/Execução
4: unknownTasks:                                  ▷ Gestor de Tarefas Desconhecidas
5: communication:                                 ▷ Gestor de Comunicação
6: dequeueWaitTimeWindow:                         ▷ Janela Temporal para a Remoção de uma Tarefa

7: function DEQUEUE
8:     position ← 0
9:     while TRUE do
10:        task ← waitToRunTasksQueue.remove(position).getTask()
11:        dequeueTimeStamp ← currentTimeStamp()
12:        communication.dequeue(task.id, new DequeueInfo(task.id, self, dequeueTimeStamp, ...))
13:        taskEntry ← locallyKnownTasksMap.get(task.id)
14:        wait for dequeueWaitTimeWindow
15:        if taskEntry.dequeueList.size() > 0 then        ▷ Vários pedidos para remoção da tarefa
16:            dequeueRanking.sort(taskEntry.dequeueList)          ▷ Sorteio de dequeueing
17:            position ← task.dequeueList.find(self)
18:            taskEntry.state ← RUNNING
19:            if position = 0 then                                ▷ Venceu o sorteio
20:                taskEntry.node ← self
21:                taskEntry.dequeueTimeStamp ← dequeueTimeStamp
22:                return task                                     ▷ Executa a tarefa localmente
23:            else
24:                DequeueInfo winner ← taskEntry.dequeueList.get(0)
25:                taskEntry.node ← winner.nodeID
26:                taskEntry.dequeueTimeStamp ← winner.dequeueTimeStamp
27:                numElementsRemoved ← 0
28:                while numElementsRemoved < position do
29:                    waitToRunTasksQueue.remove(numElementsRemoved)
30:                    numElementsRemoved ← numElementsRemoved + 1;
31:                if waitToRunTasksQueue.size > 0 then
32:                    taskEntryToExecute ← waitToRunTasksQueue.remove(0)
33:                    taskEntryToExecute.node ← self
34:                    taskEntryToExecute.dequeueTimeStamp ← dequeueTimeStamp
35:                    return taskEntryToExecute.getTask()
36:        else                                                ▷ Único pedido para remoção da tarefa
37:            taskEntry.node ← self
38:            taskEntry.dequeueTimeStamp ← dequeueTimeStamp
39:            taskEntry.state ← RUNNING
40:            return task                                     ▷ Executa a tarefa localmente

41: procedure ON_DEQUEUE(taskID, dequeueInfo)
42:     if locallyKnownTasksMap.contains_key(taskID) then   ▷ A tarefa é conhecida localmente
43:         taskEntry ← locallyKnownTasksMap.get(taskID)
44:         task ← taskEntry.task
45:         if waitToRunTasksQueue.contains(taskID) then        ▷ A tarefa está presente na fila
46:             waitToRunTasksQueue.remove(taskID)
47:             task.state ← RUNNING
48:             task.node ← dequeueInfo.nodeID
49:         else if task.state = DONE then                      ▷ A tarefa já foi executada
50:             communication.info(DONE, task.id, taskEntry.conclusionTimeStamp)
51:         else if task.state = RUNNING then                   ▷ Execução concorrente
52:             position ← dequeueRanking.sort({dequeueInfo})       ▷ Desempate pela execução
53:             if position ≠ 0 then
54:                 task.cancel()                           ▷ Cancelamento da execução local da tarefa
55:                 task.node ← dequeueInfo.nodeID
56:         else        ▷ Remoção concorrente (dois dispositivos a removerem a mesma tarefa)
57:             taskEntry.dequeueList.add(dequeueInfo)
58:     else                                                ▷ A tarefa não é conhecida localmente
59:         myStrategy ← getUnknownTasksStrategy()
60:         numTries ← 0
61:         while numTries < 5 do
62:             if myStrategy = NEVER_WANT then                 ▷ Nunca deseja obter os dados
63:                 break
64:             else if myStrategy = ALWAYS_WANT then       ▷ Deseja obter sempre os dados
65:                 unknownTasks.solveStrategy()
66:             else        ▷ Deseja obter os dados, consoante o número de dispositivos na rede
67:                 if NUM_DEVICES_ONLINE >= MIN_DEVICES_ONLINE then
68:                     unknownTasks.solveStrategy()
69:                 else
70:                     break
71:                 numTries ← numTries + 1
72:                 isKnownLocally ← locallyKnownTasksMap.contains_key(taskID)
73:                 isFinished ← finishedTasksMap.contains_key(taskID)
74:                 if isKnownLocally or isFinished then
75:                     break
```

4 Conclusões

Neste artigo apresentamos *DiCE*, uma framework para computação distribuída em dispositivos móveis com o sistema operativo Android. Esta framework permite processar e executar tarefas de forma colaborativa num ambiente de uma rede com dispositivos móveis e que pode ser volátil, com perdas de mensagens e entrada/saída de nós.

Algumas das principais funções foram implementadas, apesar de haver uma inconsistência no algoritmo *Sorteio e Desempate de Dequeuing/Execução* que não foi possível ter sido resolvida. Com base nessa inconsistência, os testes não foram suficientes para se retirar conclusões sobre a framework.

Reserva-se para trabalho futuro, a verificação de todas as situações possíveis envolvendo tarefas na fila, para confirmar e provar que essa propriedade é completamente válida para qualquer situação que possa ocorrer no sistema.

Algorithm 2 Resolução de Tarefas Desconhecidas

```
1: locallyKnownTasksMap:                          ▷ Mapa de Tarefas Localmente Conhecidas
2: waitToRunTasksQueue:                           ▷ Fila/Lista de Tarefas Prontas a Executar
3: runningTasksMap:                               ▷ Mapa de Tarefas em Execução
4: finishedTasksMap:                              ▷ Mapa de Tarefas já Executadas
5: unknownTasks:                                  ▷ Gestor de Tarefas Desconhecidas
6: communication:                                 ▷ Gestor de Comunicação
7: responseWaitTimeWindow:                        ▷ Janela Temporal para Aguardar por mais Respostas

8: function UNKNOWNTASKS.SOLVESTRATEGY(taskID)
9:     communication.doYouKnowThisTask(taskID)
10:    communication.startsListenByType(TAKE_MY_TASK)
11:    communication.startsListenByType(ALREADY_FINISHED_TASK)
12:    wait for responseWaitTimeWindow
13:    communication.stopsListenByType(TAKE_MY_TASK)
14:    communication.stopsListenByType(ALREADY_FINISHED_TASK)

15: procedure ON_DO_YOU_KNOW_THIS_TASK(taskID)
16:    onlyIAnswered ← true
17:    isKnownLocally ← locallyKnownTasksMap.contains_key(taskID)
18:    isFinished ← finishedTasksMap.contains_key(taskID)
19:    if isKnownLocally or isFinished then
20:        communication.startsListenByType(TAKE_MY_TASK) { onlyIAnswered ← false }
21:        communication.startsListenByType(ALREADY_FINISHED_TASK) { onlyIAnswered
   ← false }
22:        timeToWait ← randomNumber()
23:        wait for timeToWait
24:        communication.stopsListenByType(TAKE_MY_TASK)
25:        communication.stopsListenByType(ALREADY_FINISHED_TASK)
26:        if onlyIAnswered = true then
27:            resolveUnknownTask(taskID)

28: procedure RESOLVE_UNKNOWN_TASK(taskID)
29:    isReadyToRun ← waitToRunTasksQueue.contains_key(taskID)
30:    isKnownLocally ← locallyKnownTasksMap.contains_key(taskID)
31:    isExecuting ← runningTasksMap.contains_key(taskID)
32:    isFinished ← finishedTasksMap.contains_key(taskID)
33:    isReadyState ← isReadyToRun and isKnownLocally and not isExecuting and not isFinished
34:    isRunningState ← not isReadyToRun and isKnownLocally and isExecuting and not isFin-
   ished
35:    isFinishedState ← not isReadyToRun and not isKnownLocally and not isExecuting and
   isFinished
36:    if isReadyState then
37:        taskEntry ← locallyKnownTasksMap.get(taskID)
38:        communication.takeMyTask(taskEntry.task, taskEntry.priority, taskEntry.task.node,
   taskEntry.enqueueTimeStamp, READY)
39:
40:    else if isRunningState then
41:        taskEntry ← locallyKnownTasksMap.get(taskID)
42:        communication.takeMyTask(taskEntry.task, taskEntry.priority, taskEn-
   try.task.enqueueNode, taskEntry.enqueueTimeStamp, RUNNING)
43:
44:    else
45:        finishedTask ← finishedTasksMap.get(taskID)
46:        communication.alreadyFinishedTask(finishedTask.id, finished-
   Task.executionConcludedTimeStamp)
```