XARXES OPORTUNÍSTIQUES

OPTIMITZACIÓ DE MECANISMES D'ENCAMINAMENT

Memòria del projecte d'Enginyeria en
Informàtica realitzat per Josep Castellví
Anguera i dirigit per Ramon Martí Escalé i
Carlos Borrego Iglesias.
Bellaterra, 18 de juny de 2013

El sotasignat, Ramon Martí Escalé

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en Josep Castellví Anguera.

I per tal que consti firma la present.

Signat: Ramon Martí Escalé

Bellaterra, 18 de juny de 2013

All things are difficult before they are easy.

Thomas Fuller

Índex

Agraïments

Als meus pares, la meva família i els meus amics per haver-me aguantat durant aquest mesos. Als meus tutors, el Ramon Martí i el Carlos Borrego, per la paciència i l'ajuda tant pel que fa a la part tècnica com a la part formal del treball. A la resta de personal del dEIC que també m'han donat un cop de mà amb els aspectes tècnics quan més ho necessitava, especialment a l'Adrián Sánchez i al Rubén Martínez. Sense vosaltres aquest treball no hauria estat possible.

Capítol 1

Introducció

1.1 Motivacions

En les societats desenvolupades, estem acostumats a que, quan volem llegir la versió digital d'un diari, quan volem connectar-nos a les xarxes socials o si volem buscar el significat d'un mot a Internet podem fer-ho, sempre que disposem d'una connexió a la xarxa, d'una manera més o menys immediata. Tant si és de dia com de nit, tant si fa fred com si fa calor, faci sol o estigui plovent.

No fa falta ni que ens trobem al nostre domicili o a la feina, des d'enmig del carrer podem connectar-nos a Internet utilitzant la xarxa 3G o 4G i podem saber la informació meteorològica, enviar missatges als amics o fins i tot jugar a jocs en xarxa.

Ara bé, si ens hi parem a pensar, ens donem compte que aquesta omnipresència de la xarxa no és tant preponderant com de vegades pensem. Si anem en avió, de creuer, a la muntanya o àdhuc si ens trobem a l'interior d'un edifici moltes vegades hi ha estones en què no podem fer ús de la xarxa. A més a més, en un futur proper, l'espècie humana es començarà a establir en indrets fóra de la Terra on tampoc és possible tenir accés a Internet tal com el coneixem i en un espai proper en el temps no sembla pas possible estendre un cable de fibra òptica des de la Terra a la Lluna o fins a Mart.

Fins i tot a l'interior de les nostres ciutats, es poden donar situacions excepcionals (un atac terrorista, una inundació, una tempesta electromagnètica...) en què no sigui possible utilitzar les xarxes tradicionals. A més a més, en la majoria d'indrets del Tercer Món o dels països en vies de desenvolupament no hi ha possibilitats de connectar-se a la xarxa de cap forma i això esdevé

problemàtic perquè manté aquests llocs aïllats de la resta del món i dels avenços tecnològics.

En relació amb això, la fractura digital, que divideix el planeta entre els connectats a la xarxa i els no connectats, és un problema social i econòmic que s'ha de resoldre per aconseguir que la igualtat d'oportunitats sigui una realitat a tot el món. Per això, no ho oblidem, els que tenim una formació més gran en les TIC tenim una responsabilitat afegida per acabar amb aquesta escletxa.

Els fets enumerats anteriorment són de gran rellevància en un món permanentment connectat on les noves tecnologies i la informàtica cada vegada tenen més pes. Per solucionar aquesta problemàtica s'està treballant amb un tipus de xarxes anomenades DTN (*Delay and Disruption Tolerant Networks*) que fan possible la connectivitat fins i tot en situacions on hi ha grans endarreriments i disrupcions de xarxa amb caràcter habitual. Això per si sol és una motivació més que suficient per emprendre aquest projecte.

D'altra banda, en ser aquest un projecte vinculat a la recerca, el procés d'enfrontar-se a dificultats a les quals moltes vegades no s'ha enfrontat ningú abans ha esdevingut un punt extra de motivació que ha fet possible, en part, la realització d'aquest treball.

En relació amb això cal destacar també que en fer aquest treball en un departament que es dedica a la recerca, com el dEIC, permet familiaritzar-se, a més, amb metodologies i eines de recerca de primer nivell.

A més a més, aquest projecte ha esdevingut per nosaltres una primera presa de contacte amb els projectes de codi obert com ara la plataforma d'agents Mobile-C. Això ha esdevingut una excusa per comprendre el funcionament d'aquesta projecte que és realment interessant.

En resum, les motivacions d'aquest treball són, per una banda, la col·laboració amb un projecte de gran impacte social i, per l'altra, l'aprenentatge de noves metodologies, eines i tecnologies.

1.2 Objectius

L'objecte d'aquest projecte és aprofundir en diversos esquemes d'encaminament de xarxes DTNs (*Disruption Tolerant Networks*) des d'un punt de vista eminentment pràctic ja que, actualment, l'encaminament de les Xarxes Tolerants a Disrupcions és un dels aspectes menys desenvolupats d'aquesta tecnologia i, per tant, és un camp que resta obert.

S'han proposat multitud de solucions sobre l'encaminament de les DTNs. Una és l'*spray and wait* [13] que és amb la que se centrarà aquest projecte.

spray and wait consisteix en repartir diverses còpies d'un missatge entre diferents nodes que es poden comunicar en un moment donat. En aquest projecte volem trobar una forma de repartir-les de la forma més adequada possible tenint en compte la congestió de cada node.

Això es durà a terme afegint codi d'encaminament a uns agents mòbils que transportaran la informació d'un node a un altre. Per fer això, s'utilitzarà una plataforma d'agents mòbils sobre C anomenada Mobile-C i un dimoni de *logs* creat pel dEIC, el *prosesd*, per a l'intercanvi d'informació amb plataformes d'agents d'altres nodes.

Abans d'afegir el codi d'encaminament als agents és necessària la modificació del codi de Mobile-C, ja actualitzat anteriorment pel dEIC per tal d'afegir-hi les funcions pròpies de les DTNs, com també modificar algunes de les funcions existents perquè siguin compatibles amb els canvis.

Hi ha diferents esquemes d'implementació d'*spray and wait*, el que se segueix en aquest projecte s'anomena optimització a l'origen i és el mecanisme clàssic que consisteix en determinar els paràmetres d'encaminament en DTNs en el host d'origen.

De manera més concreta els objectius d'aquest projecte són:

- Estudiar els principals mètodes d'encaminament de DTNs
- Comprendre i modificar la plataforma Mobile-C per realitzar un encaminament més eficaç
- Comprendre i modificar el dimoni *prosesd* per intercanviar informació entre plataformes
- Estudiar i implementar el mecanisme d'*spray and wait* utilitzant l'esquema d'optimització a l'origen

1.3 Estructura

L'estructura d'aquest treball és la següent:

El primer capítol després de la introducció és l'*Estat de la Qüestió* i consisteix en una breu discussió sobre la naturalesa de les DTNs i l'estat del seu

desenvolupament en el moment present. A continuació hi ha un nou capítol, *Anàlisi i Disseny*, que comprèn el disseny a grans trets de les modificacions de la plataforma Mobile-C.

El següent capítol, *Implementació*, concreta de les modificacions descrites a l'apartat anterior així com problemes que han sorgit i com s'han esmenat. A continuació el capítol *Proves* indica breument les proves que s'han fet per avaluar la correctesa de la implementació d'aquest PFC.

Finalment es troben un capítol dedicat a les conclusions a les quals s'ha arribat amb aquest projecte, la bibliografia i un annex sobre instal·lació i posta a punt de Mobile-C en màquines virtuals.

Capítol 2

Estat de la Qüestió

En aquesta secció s'introdueix breument l'estat de la qüestió començant per exposar les limitacions de les xarxes tradicionals que ocasionen que es comencin a desenvolupar un nou tipus de xarxes, les DTNs, que es descriuen breument en el següent subapartat. Finalment es donen unes quantes pinzellades sobre l'encaminament en aquestes xarxes cosa que esdevé la base d'aquest projecte.

2.1 Limitacions de TCP/IP

Les xarxes de comunicació tradicionals com ara Internet poden ser modelades mitjançant grafs connectats en els quals sempre és possible trobar un camí entre un parell de nodes qualsevol. Els nodes es corresponen amb els dispositius connectats a la xarxa i les arestes amb les connexions que uneixen aquests dispositius. [7]

En aquest tipus de xarxes se suposa que qualsevol aresta que connecta dos nodes és bidireccional ja que suposem una amplada de banda simètrica a totes les connexions amb latència de l'ordre de microsegons i probabilitat d'error molt baixa. A més a més, suposem que en aquest tipus de xarxes l'arquitectura és més o menys estàtica en el sentit que els nodes de xarxa estan rarament desactivats.

En les xarxes tradicionals, quan les dades es troben en un node que no és el destí final, s'emmagatzemen en una memòria intermèdia temporal fins que són reenviades cap a al següent node. Com que se suposa que les dades residiran poc temps en aquesta memòria les mides d'aquests *buffers* són relativament petites.[7]

Ara bé, hi ha xarxes on no és possible fer aquesta sèrie de suposicions. Per exemple, les que estan en medis extrems com ara entre vaixells enmig de l'oceà, en zones on es desenvolupen conflictes bèl·lics, sota l'aigua, en erupcions volcàniques, a l'espai profund o també en regions en vies de desenvolupament.

El esquemes de xarxes tradicionals també poden no ser suficients en situacions com ara un atac terrorista, congestió extrema d'una xarxa o en fenòmens naturals com ara inundacions, huracans, tempestes electromagnètiques etc.

En escenaris com aquests no sempre és possible trobar un camí entre dos nodes qualssevol, els quals, a més a més, no sempre estan disponibles. En situacions com aquestes l'ample de banda esdevé asimètric i el fet de no poder reenviar el missatges en espais curts de temps fa que les memòries intermèdies hagin de ser més grans.

És més, si ens fixem concretament en el grup de protocols TCP/IP tenim un seguit de característiques d'aquests protocols que no són coherents amb aquests escenaris.

Un dels factors més determinants és que donat que els datagrames IP tenen una vida màxima de 255 segons no és possible l'ús de TCP/IP en escenaris on es pressuposen endarreriment superiors ja que d'aquesta manera es podria donar un datagrama per mort quan realment encara s'estigués enviant.

D'altra banda, les estratègies d'encaminament tradicionals del protocol no són vàlides per aquest tipus de xarxes donat que no és possible conèixer l'estructura de xarxa *a priori* perquè aquesta canvia contínuament.

2.2 Xarxes Tolerants a Endarreriments i Interrupcions

Per donar resposta a aquesta problemàtica, en un primer moment, es van desenvolupar un tipus de xarxes anomenades ICNs (*Intermittently Connected Networks*) que són xarxes sense fils que no tenen una infraestructura concreta que fan possible el funcionament d'una o més aplicacions en un medi en concret on hi ha interrupcions i endarreriments habitualment. [7]

Exemples d'aquestes xarxes són les EMNs (*Exotic Media Networks*), les WSNs (*Wireless Sensor Networks*) o les MANETs (*Mobile Ad-Hoc Networks*).

Potseriorment, com que aquestes xarxes només són d'utilitat en medis molt determinats es va decidir crear un tipues de xarxes de propòsit més general, les DTNs (*Disrution Tolerant Networks*).

Les DTNs són xarxes on és habitual que no hi hagi connectivitat permanent entre els nodes i que permeten la implementació de xarxes de comunicacions en llocs heterogenis on no seria possible mitjançant xarxes tradicionals. [7]

El desenvolupament de DTNs es va començar a tenir en compte als anys 70 quan la disminució de la mida dels ordinadors va fer possible l'aparició de xarxes amb dispositius mòbils susceptibles a grans endarreriments. Tot i això, no va ser fins a la dècada dels 90 que es va tenir un interès més gran per desenvolupar d'aquestes xarxes amb l'aparició de protocols de transmissió de dades sense fils com ara MANET.

Exemple d'escenaris

Les DTNs es poden aplicar en multitud d'escenaris, a tall d'exemple n'enumerarem alguns.

El primer exemple és la IPI (*Inter-Planetary Internet*) [2]. Consisteix en considerar l'espai exterior com un ambient caòtic on hi ha diversos regions amb internets ordinàries. L'objectiu de les IPI és interconnectar aquestes internets perquè puguin funcionar de manera conjunta. Aquest tipus d'escenaris necessiten unes especificacions lleugerament diferents de les DTNs terrestres perquè en ser les distàncies molt més grans s'han de comprendre temps de retransmissió més elevats. A més a més, es troben amb dificultats addicionals com ara temperatures dels nodes extremes (en l'espai exterior poden variar entre extremadament baixes i molt elevades) o interferències electromagnètiques que poden ocasionar que els comptadors de temps del nodes donin mesures equivocades.

Un altre exemple d'escenari és col·locar petits retransmissors a animals com en el projecte *ZebraNet* per la rastrejar fauna salvatge en experiments biològics. [6]

El projecte ZebraNet consisteix en equipar zebres amb collarets que inclouen un receptor de GPS, una memòria *flash*, transceptors sense fils i una petita CPU. El seu objectiu és la transmissió de dades sobre els moviments de zebres en el seu hàbitat als investigadors utilitzant el mínim d'energia i emmagatzematge temporal i està funcionant amb èxit al *Mpala Research Centre*, a Kenya.

Arquitectures de DTNs

L'arquitectura de les DTNs pot ser molt variada i no és necessàriament homogènia.

A continuació comentarem breument, a tall, d'exemple, una sèrie d'arquitectures que considerem representatives.

Per exemple l'arquitectura MADTN (*Multiagent Architecture for Dynamic Triage Networks*) s'utilitza per informar de l'estat de víctimes ens situacions d'emergència. [9] Això es fa mitjançant dispositius tàctils amb sensors GPS i una interfície sense cables entre els quals es mou un agent *JADE*. Aquest sistema inclou un sistema de prioritització de missatges on, per exemple, l'avís d'una visita rutinària té menys prioritat que el d'una intervenció d'urgència.

Una altra arquitectura, que s'utilitza per transmetre dades mèdiques en zones en vies de desenvolupament, col·loca sensors en vehicles de transport públic per a intercomunicar zones remotes amb els centres sanitaris. [11] Aquest és un sistema on les mides de les dades a enviar són relativament petites i es disposa d'un ample de banda limitat (en les simulacions s'utilitzen transmissors-receptors *bluetooth*).

Un exemple d'arquitectura molt interessant és utilitzar els dispositius de telefonia mòbil com a nodes d'una DTN quan ens trobem en una zona sense accés en Internet, per exemple en un tren que passa per un túnel. Per exemple, *Haggle*, [10] utilitza dispositius de telefonia mòbil per, quan es troba amb un veí, quan es generen noves dades o cada un cert interval de temps, intercanviar dades.

Finalment, l'arquitectura més utilitzada és la basada en els anomenats protocols *bundle* que es comenta amb un xic més de detall a la següent subsecció.

Els Protocols *Bundle*

Tot i que en aquest projecte no hem utilitzat cap protocol específic de DTNs ja que aquest fet és pràcticament independent de l'encaminament és necessari descriure breument, en aquest apartat, el protocol que utilitzen la majoria de les DTNs. S'anomena protocol *bundle* i divideixen les dades a enviar en diversos *bundles* (fardells) que contenen informació sobre el seu encaminament.

El protocol consisteix en una sèrie de nodes que porten petits sensors els quals s'intercanvien *bundles* i poden demanar que els n'enviïn mitjançant un tipus especial de missatge anomenat *beacon*.

El protocol *bundle* actua com una nova capa de la pila de protocols de comunicació entre la capa de transport i la d'aplicació del model de capes de TCP com es mostra en la figura 2.1. Es pot donar el cas que la xarxa que hi ha per sota del protocol *bundle* no implementi aquest model de capes ja que això depèn del tipus de cada xarxa però el protocol *bundle* es trobarà a un nivell equivalent.

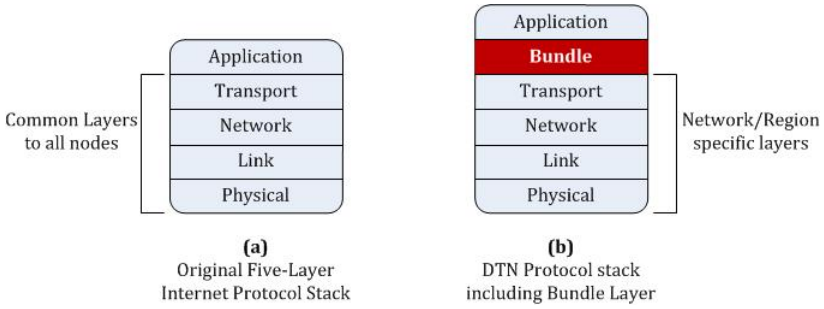

Figura 2.1: Inclusió del protocol *bundle* en el model de capes de TCP.

Cada node que participa en el protocol *bundle* està identificat per un EID (*Endpoint IDentifier*) que té un màxim de 1024 bytes i que pot veure's com un URI (*Universal Resource Locator*). També podem tenir un EID que es refereixi a un grup de nodes (Això és útil per la implementació del *multicast*). El nom de l'esquema és *dtn:* i es posa *dtn:none* quan es vol excloure qualsevol adreça. En els altres casos la part jeràrquica que ve a continuació encara és objecte de discussió i no s'ha arribat a una conclusió clara.

En tot cas, el fet que s'utilitzin URIs en lloc d'adreces és conseqüència que es vol identificar un node o un grup de nodes en concret i no una localització determinada perquè se suposa que són nodes mòbils i, per tant, és útil referir-nos a un recurs concert i no a una certa localització. [7]

El protocol *bundle* s'ocupa també d'implementar el control de temps però d'una forma poc activa. Bàsicament dóna per fet que tan els extrems com els nodes intermedis d'una comunicació estan sincronitzats en el temps. Actualment s'estan investigant mecanismes perquè es pugui assegurar aquest fet. [7]

Un dels aspectes més sensibles del protocol *bundle* és la gestió de les custòdies. En TCP la capa de transport comporta que els nodes intermedis per on han de passar les dades esdevinguin transparents per l'emissor i el receptor i fa que aquests extrems puguin suposar que la transmissió s'ha dut a terme sense errors. Això no és possible en xarxes DTN perquè tenint en compte que els temps de transmissió acostumen a ser elevats no és possible guardar les dades en *buffers* intermedis per esperar la confirmació per falta d'espai en aquestes memòries temporals.

Per solucionar això el protocol *bundle* introdueix un sistema de custòdies mitjançant el qual un node traspassa la custòdia d'un fardell al següent node el qual assumeix la responsabilitat d'enviar-lo al seu destí.

Podem descriure el procés de transferència de custòdia de la següent manera: Suposem que un node qualsevol té la custòdia d'un fardell concret. Mentre aquest node no està en contacte amb cap veí manté el fardell desat en una memòria. Quan troba un veí li envia el missatge i aquest l'emmagatzema a la seva memòria. Llavors, el primer node comprova que l'altre compleixi els requisits per fer-se càrrec del fardell i, si és així, li envia una sol·licitud dient-li si vol acceptar la custòdia del fardell. Si aquest el contesta afirmativament abans d'un interval de temps determinat el primer node pot eliminar el fardell de la seva memòria temporal. En cas contrari, tornarà a enviar sol·licitud de custòdia fins que rebi una resposta afirmativa o li pugui passar la custòdia a un altre veí (figura 2.2).

Aquest sistema de transferència de la custòdia té una sèrie de punts febles que fan que no pugui tenir la mateixa funcionalitat que la capa de transport a TCP/IP.

El primer dels quals és que no inclou cap sistema per detectar o corregir errors en un fardell. Un altra problema és que en el mecanisme descrit se suposa que tenim una comunicació bidireccional (Un node envia una sol·licitud de custòdia i l'altre envia una confirmació al primer cas d'haver-la acceptat.). Això no és així moltes vegades en DTNs perquè sovint l'origen és únicament un transmissor.

Un altre punt feble d'aquest mecanisme és que es basa en què un node té la voluntat d'acceptar la custòdia dels fardells però això no és així si aquest node està gestionat de manera egoista de forma que processi el menor nombre de dades. Això depèn únicament del node en qüestió. Una proposta per esquivar aquest problema podria ser enviar uns fardells especials anomenats *reports* a un cert node que s'anomenaria *report-to* i que coordinaria la gestió de les custòdies, entre d'altres funcions.

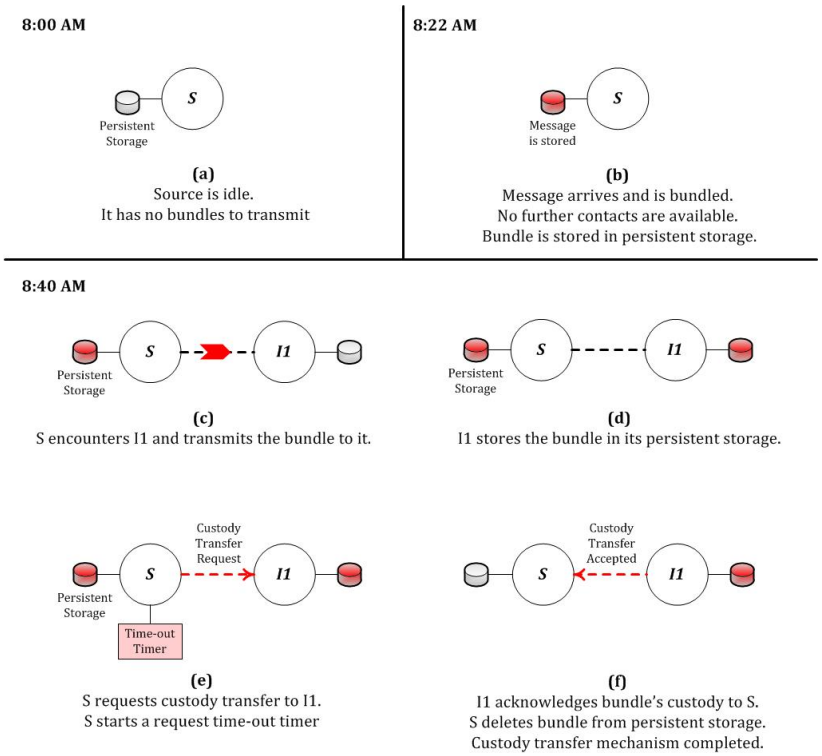

Figura 2.2: Procés de transferència de la custòdia en el protocol *bundle*. En aquest exemple el node S transfereix la custòdia al node I1. [7]

Pel que fa al control de la congestió en xarxes DTNs hi ha més problemes que en les xarxes tradicionals perquè es poden produir grans acumulacions de fardells en els nodes. Per solucionar-ho s'ha proposat el fet d'assignar prioritats als fardells. Els fardells en custòdia tenen una prioritat més elevada que els que no en tenen i els fardells petits són més prioritaris que els grans.

Tot i això aquest sistema no sempre funciona satisfactòriament i és necessari trobar sortides a les situacions en les quals no ens és útil.

Com que una DTN pot comprendre xarxes físiques heterogènies és necessari establir un sistema per gestionar la fragmentació de fardells.

Fins al dia d'avui s'han estudiat dues propostes. La primera és la *fragmentació proactiva* que es decanta per fragmentar els fardells en el mode d'origen de manera que siguin el suficient petits com per ser compatibles amb totes les infraestructures amb que es puguin trobar. En canvi, la sego-

na s'anomena *fragmentació reactiva* i fragmenta els fardells quan es troba amb una xarxa per la qual són massa grans.

2.3 Encaminament en DTNs

Un dels problemes més grans de les xarxes DTN és la dificultat de trobar un sistema d'encaminament eficaç pels missatges donat que no és possible conèixer *a priori* l'estructura de la xarxa i, a més a més, aquesta canvia constantment. [3]

En el moment present, no hi ha cap esquema d'encaminament que funcioni satisfactòriament de manera general i aquesta és, segurament, la problemàtica més destacada de les xarxes DTN.

Algunes propostes [14] es basen en tenir un conjunt de polítiques dinàmiques per escollir entre diferents possibilitats d'encaminament. En la capçalera dels paquets hi ha un identificador que indica a quin flux pertany i determina com es farà el seu encaminament. [14]

El primer d'aquests bits ens indica si estem davant d'un paquet de dades (0) o d'un paquet de control (1). El segon bit, ens indica com d'important és el temps per l'aplicació que envia les dades. Si és "0"estem davant d'una aplicació que no és en temps real i si és "1"si que ho és. Per últim, el tercer bit ens indica si no s'exigeix fiabilitat en la transmissió de les dades (0) o si sí que se n'exigeix (1).

A partir d'aquests bits els paquets es col·loquen en una cua més o menys prioritària i, d'aquesta manera, s'aconseguix un sistema d'encaminament satisfactori en molts casos.

Tot i això, aquest model no és suficientment genèric ni flexible per ser utilitzat per propòsit general.

Una altra aproximació utilitza una funció amb paràmetres com ara la mida del missatge, el TTL (*Time To Live*) o el nombre de nodes a la xarxa [12] per calcular els paràmetres de replicació. El problema aquí és que els paràmetres es determinen de forma estàtica i no es té en compte la coexistència de més d'un escenari. A més a més, com en el cas de l'*spray and wait*, moltes vegades no podem saber, *a priori*, el nombre de nodes en una xarxa DTN.

D'altres propostes incorporen l'ús d'algorismes concrets com ara el PROPHET [8] o el MaxPROP [4] per estimar els paràmetres amb què es dissemi-

naran els missatges. També incorporen el concepte d'escales temporals ja que argumenten que si els missatges tenen un TTL petit el comportament dels mecanismes d'encaminament és més determinant que si tenen un temps de vida més llarg.

Aquestes aproximacions, tot i que demostren que una parametrització adequada és important, segueixen sense tenir en compte més d'una aplicació en el mateix escenari.

Algunes altres propostes tenen en compte el control del flux de dades node a node i la reducció de l'ample de banda dels nodes que realitzen més reenviaments però cap contempla escenaris multi-aplicació on es tingui en compte l'estat de la xarxa en cada moment.

D'altres propostes es basen en la tècnica d'*spray and wait*. L'*spray and wait* (escampar i esperar) consisteix en dues fases: [13]

La primera fase és la fase d'*spray* i consisteix en repartir entre els nodes veïns i el mateix node origen un total de L còpies que hi haurà a la xarxa. Aquestes còpies s'aniran distribuint entre un màxim de L veïns en total. La segona fase és la fase d'espera, o *wait*. En aquesta fase, cas que no s'hagi trobat el destí en l'*spray* es fa transmissió directa dels L missatges.

L'aspecte més delicat d'aquest mètode és la necessitat d'escollir la manera com calcular el paràmetre L i escollir com distribuir les L còpies.

Pel que fa al càlcul del paràmetre L depèn únicament del nombre de nodes d'una xarxa, M, i d'un factor, a, que ens indica el retard màxim que podem acceptar en funció del nombre de vegades el retard màxim.

La fórmula de càlcul és la següent: [13]

$$(H_M^3 - 1.2)L^3 + \left(H_M^2 - \tfrac{\pi^2}{6}\right) L^2 + \left(a + \tfrac{2M-1}{M(M-1)}\right) L = \tfrac{M}{M-1}$$

on $H_n^r = \sum_{i=1}^{n} \frac{1}{i^r}$ és el n$_{\text{è}}$ nombre harmònic d'ordre r.

Tot i això, normalment, en les DTNs, ens serà impossible saber el nombre de nodes de la xarxa. Malgrat que hi ha fórmules d'estimació de L sense necessitat de saber el nombre de nodes de la xarxa els resultats no sempre són el suficientment precisos.

Pel que fa a la manera d'escollir la forma de distribuir les còpies una aproximació que funciona bé és l'*spray and wait* binari. Aquest mecanisme

consisteix en què quan un node amb ($n > 1$) còpies se'n troba un altre li passa ($n/2$) còpies i se'n queda ($n/2$) per a ell mateix.

Aquest plantejament és òptim quan el moviment dels nodes és independent i idènticament distribuït (IID) però no ho és en la resta de casos. A més a més, no té en compte la congestió.

Per tant, en l'actualitat no hi ha cap mecanisme d'encaminament que sigui satisfactori en tots els casos i, per tant, aquest és un espai de recerca totalment obert.

Capítol 3

Anàlisi i Disseny

3.1 Introducció

El marc d'aquest treball és estudiar i implementar el mecanisme de l'*spray and wait* on els paràmetres de replicació i el número de còpies a enviar es determinen de forma dinàmica a partir de la congestió dels nodes de la xarxa. Això es fa introduint codi d'encaminament en els missatges que s'envien.

Per determinar els paràmetres de l'*spray and wait* de forma dinàmica hi ha quatre mètodes:

El primer és l'*optimització a l'origen* que consisteix en determinar els paràmetres d'encaminament en DTNs en el host d'origen.

Un altre mecanisme és l'*optimització al node* que determina els paràmetres d'encaminament de forma dinàmica en a cada un dels nodes.

En el *control de switching* es determinen els paràmetres òptims a partir de la realització de simulacions d'escenaris aleatoris. En canvi, el *gain scheduling control* determina els paràmetres d'encaminament a partir de la interpolació de simulacions dels diversos paràmetres per un cert rang de valors.

En l'actualitat, una de les àrees de recerca més actives en DTNs és desenvolupar aquests mecanismes i, de fet, aquest és el marc d'aquest treball en què implementarem el mecanisme d'optimització a l'origen.

Des d'un punt de vista més pràctic, utilitzarem agents mòbils que transportaran missatges d'una plataforma d'agents a l'altra. La plataforma que utilitzarem serà Mobile-C (versió 2.1.3), amb modificacions del dEIC per

treballar amb DTNs, i per a l'intercanvi d'informació utilitzarem un dimoni, desenvolupat pel dEIC, el *prosesd*.

En aquest capítol descriurem breument els canvis fets en el codi de Mobile-C, en el *prosesd* i el codi d'encaminament que hem afegit en els agents.

Es tracta d'introduir codi al agents per que, quan es trobin al node origen, calculin el valor de L, és a dir de les còpies dels missatges que hi haurà a la xarxa i, després, mitjançant *prosesd*, busquin quins veïns tenen al seu abast i n'examinin la congestió. A partir d'aquí les còpies s'enviaran als diferents veïns de forma inversament proporcional a la seva congestió.

Cal remarcar que si s'ha d'enviar més d'una còpia a un node no s'hi enviarà més d'un agent sinó un sol agent amb un paràmetre L que indicarà el nombre de còpies que representa ja que no tindria sentit enviar dos agents idèntics al mateix node.

L'objecte d'aquest projecte ha estat introduir codi d'encaminament en els agents per implementar el mecanisme descrit. Ara bé, per fer-ho s'ha hagut de modificar també Mobile-C i *prosesd*. Per tant, a les següents seccions, farem una breu descripció del seu funcionament i dels canvis que s'hi faran. Un esquema general es mostra a la figura 3.1.

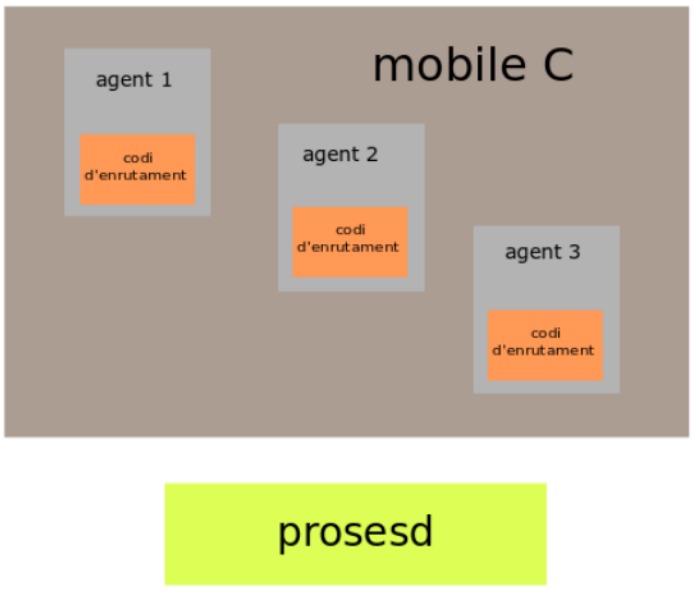

Figura 3.1: Esquema general dels elements afectats pel present projecte.

3.2 Estructura i canvis a Mobile-C

Mobile-C és una llibreria que fa possible introduir una agència dins d'un programa. Una agència és una plataforma d'agents mòbils en la qual aquests existeixen i operen. Mobile-C té una API amb funcions per controlar l'agència i els agents que hi ha.

FIPA (*Foundation for Intelligent Physical Agents*) és una organització pertanyent a IEEE que s'encarrega d'assegurar la interoperabilitat entre les diferents plataformes d'agents mòbils.

Mobile-C està dissenyada seguint les recomanacions de FIPA i, per això, està formada pels següents mòduls que s'executen en *threads* separats (figures 3.2 i 3.3):

- *Agent Management System* (AMS): S'encarrega de la creació, registre, migració, persistència i interrupció de l'execució dels agents.
- *Agent Communication Channel* (ACC): S'encarrega de la comunicació entre els agents i del transport d'agents entre una plataforma i una altra.
- *Directory Facilitator* (DF): Serveix de directori de pàgines grogues.
- *Agent Execution Engine* (AEE): És un entorn d'execució multiplataforma per llançar els agents.
- *Agent Security Manager* (ASM): Inclou seguretat a Mobile-C (identificació d'usuaris, autentificació, autorització d'agents, etc.).

Per últim, l'arxiu *libmc.c*, conté la implementació de les funcions de la llibreria Mobile-C i una sèrie d'arxius: *agent.c*, *agent_datastate.c* i *agent_task.c* que defineixen els agents.

Nosaltres hem modificat l'*Agent Management System*, l'*Agent Communication Channel* i aquests tres últims arxius.

Bàsicament les modificacions efectuades s'encarreguen de fer possible la còpia d'agents des del codi d'encaminament.

En el codi original no era possible copiar un agent des del codi d'aquest propi agent ja que s'havia de bloquejar. Així, hem modificat la plataforma perquè des del codi d'encaminament es pugui indicar a la plataforma (que sí que és capaç de copiar l'agent) que cloni l'agent i li assigni una determinada destinació i un certs paràmetres d'*spray and wait*.

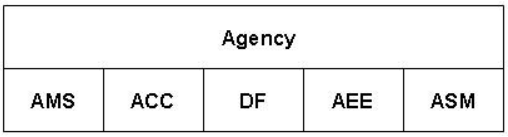

Figura 3.2: Estructura d'una agència amb cada uns dels *threads* que comprèn.

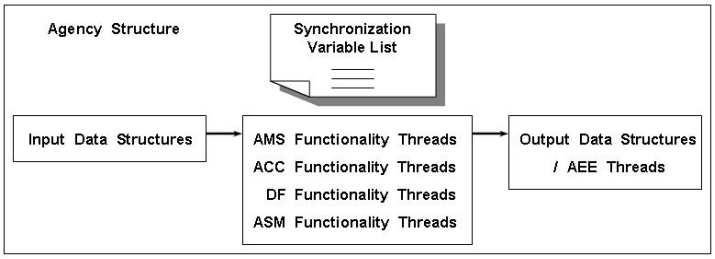

Figura 3.3: Implementació de Mobile-C

3.3 Estructura i canvis a *prosesd*

prosesd és un dimoni que s'encarrega de la comunicació entre diferents plataformes d'agents. El *prosesd*, quan la plataforma d'agents activa l'opció de descobriment de veïns, rep d'un arxiu de la plataforma (*ams.c*) una sol·licitud de descobriment i envia petits paquets de dades anomenats *beacons* a un grup *multicast* al qual estan subscrits els *prosesds* de tots els nodes. D'aquesta manera es dóna a conéixer i pot saber quins nodes estan a l'abast d'un node.

Nosaltres el modificarem perquè rebi de la plataforma un camp amb el seu nivell de congestió i l'afegeixi a una ontologia perquè la resta de plataformes en tinguin accés.

3.4 Implementació de l'algorisme d'encaminament *spray and wait* a Mobile-C

El primer que necessitem per implementar l'*spray and wait* és un mecanisme per a copiar agent. Mobile-C té una funció per copiar un agent però en certes ocasions no funcionava satisfactòriament. Per això l'hem modificat

per a que no sigui així. Els canvis a nivell de codi estan inclosos en el següent capítol.

Ara bé, un cop aquesta funció ha funcionat correctament en tots els casos ens hem adonat que no la podem utilitzar per copiar un agent en execució, ja que per a copiar un agent és necessari bloquejar-lo i un agent no es pot bloquejar a ell mateix. Per això utilitzarem una altra estratègia.

Es tracta de, des del codi d'encaminament, no copiar directament sinó *deixar dit* a la plataforma a la qual pertany l'agent que volem que ens el copiï. Aquesta, en l'arxiu *ams.c*, utilitzarà *agent_Copy* per copiar l'agent.

A més a més, hem afegit una estructura a l'estat temporal d'un agent (*datastate*) que serveix per emmagatzemar una sèrie de adreces que es corresponen amb les destinacions a què volem enviar els agents. Hem fet el mateix amb el paràmetre L que es correspon amb cada agent.

A la plataforma, un cop s'hagi fet el nombre de còpies necessàries de cada agent tindrem n agents idèntics. Aquí se'ls assigna a cada un el destí i la L que els correspon a partir de les estructures que hem anomenat en el paràgraf anterior.

Es calcula la L de manera que aquesta sigui més gran com més congestionat estigui l'origen respecte els seus veïns. Això és així perquè si l'origen té un grau de congestió substancialment més gran que els seus veïns se suposa que aquest es vol treure els agents de sobre i si és a l'inrevés preferix no congestionar més veïns i quedar-se ell els agents en espera d'una situació més favorable.

Un esquema general es mostra a la figura 3.4:

La resta de la implementació de l'*spray and wait* s'inclou en el codi d'encaminament dels agents.

3.5 Planificació Temporal

Pel que fa a la planificació inicial, consistia en implemetar els quatre mecanismes d'optimització d'*spray and wait*: *optimització a l'origen*, *switching control*, *scheduling control* i *scheduling control*. Mentrestant havíem planificat la redacció de la memòria i, més tard, la revisió de la mateixa. Finalment hem vist que per qüestions de temps no era possible aprofundir en els quatre mecanismes i, per tant, ens hem centrat en el primer, l'optimització a l'origen.

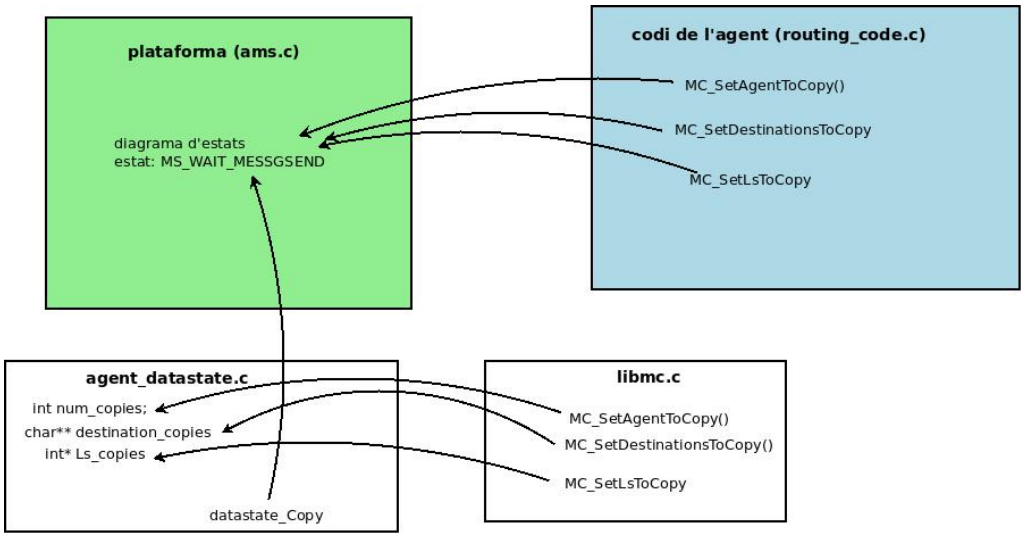

Figura 3.4: Diagrama conceptual de la còpia d'un agent

El diagrama de Gantt de la planificació inicial es mostra a la figura 3.5.

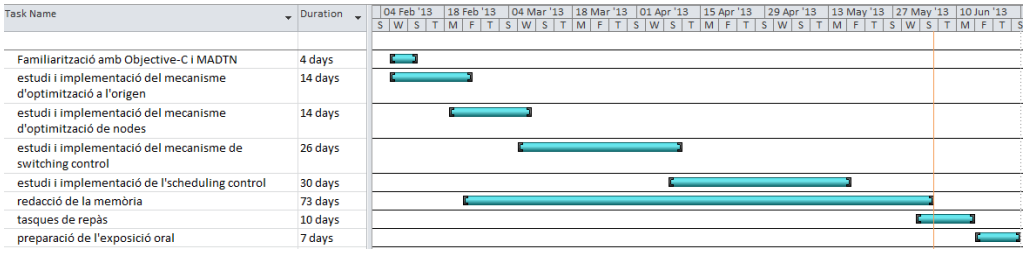

Figura 3.5: Diagrama de Gantt de la planificació temporal original

3.6 Estudi de Viabilitat

Viabilitat Tècnica

Pel que fa a la viabilitat tècnica ja hi ha hagut experiments que han demostrat la viabilitat tècnica de les DTNs com ara DINET (*Deep Impact Networking*) de la NASA o l'experimentació que està duent a terme actualment el SENDA (*Security of Networks and Distributed Applications*) en escenari aeronàutics.

A més a més, mecanismes com ara *spray and wait* ens permetran treballar amb múltiples còpies d'un mateix missatge que, combinats amb tècniques

d'encaminament adients, ens permetran afegir redundància als missatges i augmentar la probabilitat de retransmissió amb èxit en escenaris susceptibles a grans endarreriments o disrupcions.

Si ens fixem en la seguretat, hi ha diversos models de seguretat molt prometedors com ara l'ús de PKIs (*Public-key infrastructure*) o les autoritats de certificació distribuïdes.

Viabilitat Econòmica

Aquest projecte serà viable perquè el desenvolupament d'algorismes de routing per a DTNs és un camp obert i actiu amb molt de treball per fer i té multitud d'aplicacions.

Per exemple, les comunicacions en escenaris aeris o en l'espai tindran un pes econòmic cada vegada més gran i és possible que sigui de menester mecanismes de comunicació en zones rurals de països en vies de desenvolupament on no hi ha implementat accés a Internet.

A més a més el software necessari per a la seva implementació té un cost relativament baix (no superior als 200 €).

Pel que fa al cost d'implementació, segons [5] el salari d'un programador *júnior* a Espanya és, de mitjana, 10.65 €/h. Hem invertit una mitjana de 5 hores diàries a aquest treball durant 5 dies a la setmana durant un període de 5 mesos, això són 500 hores i, per tant un cost de 5325 €.

Viabilitat Legal

Pel que fa al punt de vista legal cal destacar que l'anàlisi de les DTNs depèn dels propòsits pels qual s'utilitzin i de quines aplicacions les facin servir.

Per moltes aplicacions que utilitzen DTNs, l'autenticitat i la privadesa són cabdals i, en aquest casos, és necessari comptar amb una base legal amb la qual determinar què es pot fer i què no.

Algunes DTNs com ara les aplicacions en escenaris aeronàutics poden estar distribuides entre més d'un àmbit legislatiu i això dificulta l'establiment d'un marc legal concret.

És més, les DTNs en l'espai (entre satèl·lits, o, en un futur, en altres planetes) estan regulades per la llei de l'espai que consisteix en diversos tractats legals com ara l'*Outer Space Treaty* (1967) o el *Moon Treaty* (1979) així com legislació pròpia d'alguns països.

A l'Estat aquestea legislació està recollida, bàsicament, en dues lleis: La LOPD (*Ley Orgánica 15/1999 de 13 de diciembre de Protección de Datos de Carácter Personal*) i la LSSICE (*Ley 34/2002, de 11 de julio, de servicios de la sociedad de la información y de comercio electrónico*). La LOPD s'encarrega fonamentalment de tractar les dades de caràcter personal independentment del suport on estiguin i dictamina els drets que tenen les persones físiques sobre les seves dades personals.

L'usuari té dret d'accés, rectificació, cancel·lació i oposició de les seves dades personals que no es poden facilitar a terceres persones sense l'autorització explícita de l'afectat. Això té com a conseqüència la necessitat d'adoptar mesures en la protecció de dades personals en les DTNs.

D'altra banda, l'LSSICE regula les obligacions dels prestadors de serveis que actuïn com a intermediaris en la prestació de serveis de continguts per les xarxes de comunicació, les comunicacions industrials per via electrònica i els contractes electrònics. Com a conseqüència l'haurem de considerar si prestem serveis utilitzant DTNs.

D'altra banda, des d'un punt de vista genèric de les DTNs, el projecte plantejat és totalment viable ja que actualment no hi ha cap legislació que reguli les DTNs.

Viabilitat Operativa

Pel que fa a la viabilitat operativa la part pràctica del projecte consisteix en implementar algorismes d'encaminament mitjançant Mobile-C que és una plataforma expressament dissenyada per la programació d'agents mòbils utilitzant una extensió de Mobile-C concebuda precisament per programar DTNs.

Tenint en compte que mitjançant agents de Mobile-C, *prosesd*, i un conjunt de màquines virtuals podem modelar completament el funcionament d'una DTN i que Mobile-C i *prosesd* són projectes de codi obert i, per tant, modificables podem afirmar que aquest projecte és viable operativament.

Capítol 4

Implementació

En aquest capítol descriurem el codi que hem modificat tant a la plataforma Mobile-C, a *prosesd* com al codi d'encaminament dels agents.

Cal recordar que tot el codi que es mostra en aquesta secció s'ha implementat per aquest PFC llevat que s'indiqui el contrari.

4.1 Adaptació de Mobile-C

En aquest secció exposarem la descripció dels canvis que hem realitzat en el codi de Mobile-C juntament amb el codi més representatiu.

L'objecte d'aquestes modificacions és proporcionar funcions que permetin la funcionalitat desitjada al codi d'encaminament que s'afegirà als agents. També ha estat necessari solucionar una sèrie de mancances en codi original que impedien realitzar els canvis desitjats. De fet, aquest fet ha ocasionat una càrrega addicional d'esforç en el treball perquè s'ha hagut d'inspeccionar bona part del codi de Mobile-C.

Agent Management System (AMS)

Aquí descriurem els canvis fets a AMS que, com s'ha indicat anteriorment, s'encarrega de la creació, registre, migració, persistència i interrupció de l'execució dels agents.

ams.c

ams.c és l'arxiu principial de l'*Agent Management System* i conté la màquina d'estats que comprèn els diferents estats d'un agent i també una altra màquina virtual que controla la interacció amb *prosesd*.

còpia d'agents a la plataforma

A *ams.c* hi ha la màquina d'estats que s'encarrega de gestionar cada un dels estats en què pot estar un agent. Dins de l'estat de migració/execució hem afegit un codi perquè, quan el nombre de copies que es vulgui fer d'un agent sigui una o més, es copiï l'agent.

Primer de tot es declara un *array*, *cp*, d'agents on guardarem els agents que copiem. A continuació utilitzem la funció *agent_Copy()* per copiar l'agent actual i guardar-lo a *cp*. A continuació canviem l'estat de l'agent a *MC_WAIT_MESSGSEND* perquè es torni a tractar l'agent.

A continuació assignem una identificació única a l'agent perquè si no ho féssim tindrem dos agents amb una identificació idèntica. A partir d'aquí indiquem el pròxim salt a fer per cada un dels agents que s'han copiat i també un paràmetre L (nombre de còpies) d'*spray and wait*. Tant els següents salts com els diferents valors de L s'hauran calculat al codi d'encaminament que hem afegit a l'agent.

A partir d'aquí canviem el nom dels nous agents perquè no es doni el cas que tots els agents copiats tinguin el mateix nom i, finalment, els afegim a la llista d'agents de la plataforma, *alist*, per a que siguin tractats.

Per acabar, cas que l'agent s'hagi copiat es posa la variable que indica el nombre de còpies a fer a -1 per indicar que no volem realitzar l'operació dues vegades.

```
MCAgent_t cp[MAX_AGENTS];

int j = 0;
int id = 0;
char b[3];

printf("num copies: %d\n", current_agent->datastate->num_copies);
for (j=0; j<current_agent->datastate->num_copies; j++){
  cp[j] = agent_Copy(current_agent);
  cp[j]->agent_status = MC_WAIT_MESSGSEND;
  cp[j]->id = current_agent->id+j+1;
  if (current_agent->datastate->num_destination_copies > j){
     printf("updating destinations and Ls...\n");
```

```
      strcpy(cp[j]->datastate->destination, current_agent->datastate->
          destination_copies[j]);
      cp[j]->L = current_agent->datastate->L_copies[j];
   }
   printf("destination: %s\n", cp[j]->datastate->destination);
   printf("L of the copy: %d\n", cp[j]->L);
   cp[j]->name = (char*)realloc(cp[j]->name, sizeof(char) * (strlen(cp
       [j]->name) + 4));
   sprintf(b, "%d", j);
   strcat(cp[j]->name, b);
   agent_queue_Add(alist, cp[j]);
}

if (current_agent->datastate->num_copies > 0){
   current_agent->datastate->num_copies = -1; // Don't copy the same
       agent again.
}

          printf("# dests: %d\n", current_agent->datastate->
              num_destination_copies);

for (j=0; j<current_agent->datastate->num_destination_copies; j++){
          free(current_agent->datastate->destination_copies[j]);
}

          if (current_agent->datastate->num_destination_copies > 0){
   free(current_agent->datastate->destination_copies);
}

current_agent->datastate->num_destination_copies = 0;
```

enviament de la congestió a prosesd

També a *ams.c* hi ha una màquina d'estats de descobriment de veïns. Entre d'altres coses, aquesta màquina envia sol·licituds al dimoni de descobriment de veïns d'altres nodes per obtenir informació sobre aquests veïns.

La modificació d'aquesta màquina consisteix en afegir un valor a aquestes sol·licituds que consisteix en el nivell de congestió de la plataforma que considerem la mida de la cua d'agents per tractar de la plataforma. Hem de tenir en compte que per fer-ho haurem d'afegir un nou camp a l'estructura de les sol·licituds (*cmd_hdr*) a *ams.h*. Un cop hem afegit el valor de la congestió a la sol·licitud fem possible que els altres nodes sàpiguin quina congestió hi ha a la nostra plataforma i nosaltres puguem saber el seu nivell de congestió puix que els altres nodes també disposaran d'aquest mecanisme.

El codi afegit és el següent:

```
request.congestion = ams->mc_platform->agent_queue->size;
```

Agent Communication Channel (ACC)

L'Agent Cummication Channel s'encarrega de la comunicació entre els agents i del transport d'agents entre una plataforma i una altra.

Nosaltres hem modificat aquest últim aspecte del transport perquè ja que hem modificat l'estructura dels agents haurem de modificar aquesta part perquè actuï congruentment amb aquests canvis.

Concretament modificarem dos fitxers, *xml_compose.c* i *xml_parser.c* que s'encarreguem respectivament de convertir l'estructura de dades amb la qual es representen els agents a Mobile-C a XML per poder ser transportats i de convertir aquest XML a l'estructura original un cop arribats al destí.

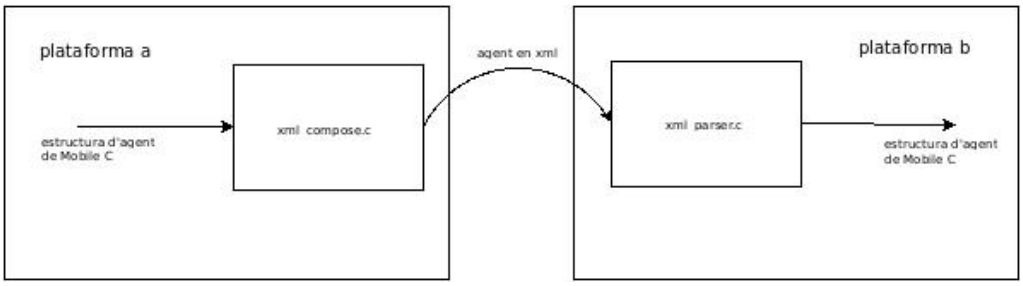

Figura 4.1: Els arxius *xml_compose.c* i *xml_parser.c*

xml_compose.c

En l'*xml_compose.c*, el primer que hem fet ha estat implementar una funció que afegeixi el paràmetre *L* d'*spray and wait* a l'XML de l'agent. Per implementar aquesta funció ens hem fixat en funcions similars que hi havia en el codi original.

Aquesta funció és *agent_xml_compose__L()* que pren com a argument un agent, n'extreu el paràmetre *L* i l'afegeix en un node XML. Per fer-ho utilitzem una extensió d'XML anomenada MXML (versió 2.2.2) per poder treballar amb fitxers XML de forma més flexible.

```
mxml_node_t*
agent_xml_compose__L(agent_p agent)
{
  mxml_node_t* node;
  char* buffer = (char*) malloc (sizeof(char)*10);
  snprintf (buffer, 10, "%d", agent->L);
  node = mxmlNewElement(
       NULL,
       "L"
       );
  mxmlNewText(
       node,
       0,
       buffer
       );
  return node;
}
```

A continuació, cridarem aquesta funció per afegir el paràmetre *L* a l'*XML* de l'agent.

```
tmp_node = agent_xml_compose__L(agent);
```

xml_parser.c

En l'*xml_parser.c* s'han fet els següents canvis:

correció d'un codi que pot donar errors a agent_xml_parse()

A aquesta funció hi hem afegit una condició a l'*if* . Només es parsejarà l'agent cas que *agent_xml_parsemobile_agent()* s'hagi executat satisfactòriament ja que si no no té sentit fer-ho i es provocaria un error de segmentació ja que s'intentaria parsejar un punter a *NULL*.

```
if (agent->datastate->is_routable == 1 && status == MC_SUCCESS) ...
```

També la funció anterior ha de retornar un valor o un altre depenent de si la funció s'ha executat correctament. Per això hem canviat la línia del codi original:

```
return MC_SUCCESS;
```

... per

```
return status;
```

parseig de L

De la mateixa manera que a *xml_compose.c* s'ha afegit el paràmetre *L* a l'XML a *xml_parser.c* haurem d'extreure'n el valor de XML i assignar-lo a l'estructura que utilitzem per representar agents en Mobile-C.

Primer que tot implementem la funció *agent_xml_parse__L()* que s'encarrega d'extreure el paràmetre L de l'XML i afegir-lo a l'estructura *agent_p* que li passem com a paràmetre.

```
error_code_t
agent_xml_parse__L(agent_p agent, xml_parser_p xml_parser)
{
  char *text;
  const mxml_node_t* L_node;

  L_node = xml_parser->node;

  text = xml_get_text( L_node );
  CHECK_NULL(text, agent->L=NULL;return MC_SUCCESS;);
  agent->L = atoi(text);
  free(text);
  return MC_SUCCESS;
}
```

A continuació cridem aquesta funció per fer efectiu el parseig del paràmetre *L* passant-li l'agent al qual volem afegir *L* i el node XML que el conté que haurem obtingut mitjançant la funció *xml_get_child()*.

```
xml_parser.node = xml_get_child(
        xml_parser.root,
        "L",
         1);
agent_xml_parse__L(agent, &xml_parser);
```

Funcions de Llibreria

Aquí parlarem de la implementació d'algunes funcions de llibreria de Mobile-C que seran les que es cridaran des del codi d'encaminament ja que seran necessàries per la implementació d'*spray and wait*.

libmc.c

libmc.c és l'arxiu que conté la definició de totes les funcions de llibreria de Mobile-C. En alguns casos ens ha estat molt útil observar com estaven implementades les ja existents per crear les noves. Hem creat les següents funcions:

MC_SetAgentToCopy i *MC_SetAgentToCopy_chdl()*

MC_SetAgentToCopy() s'encarrega d'afegir a l'estat, *datastate*, de cada agent el nombre de copies que es vol que es faci d'ell tret que l'agent que se li passi sigui la còpia d'un altre agent. En aquest cas s'haurà posat el valor del nombre de còpies que conté el *datastate* a *-1* per indicar que es tracta d'un agent que ja s'ha copiat i que no volem copiar una altra vegada. *agent* és l'agent que es vol copiar, i *num_copies* el nombre de copies que se'n vol fer. La funció retorna MC_SUCCESS cas que s'hagi executat sense errors.

EXPORTMC ens indica que aquesta funció és de Mobile-C.

Pel que fa a *MC_SetAgentToCopy_chdl()* és una funció que serveix perquè *MC_SetAgentToCopy()* es pugui cridar des de l'espai d'*script*.

El codi d'aquestes funcions és el següent:

```
EXPORTMC int MC_SetAgentToCopy(MCAgent_t agent, int num_copies)
{
        MUTEX_LOCK(agent->lock);

        /* Number of times the agent to be copied */
        if (agent->datastate->num_copies != -1){   // Don't copy the
            agent more than a time.
          agent->datastate->num_copies = num_copies;
        }
        else{
          agent->datastate->num_copies = 0;
        }

        MUTEX_UNLOCK(agent->lock);

        return MC_SUCCESS;
}
```

```
EXPORTCH int MC_SetAgentToCopy_chdl(void *varg)
{
        MCAgent_t agent;
        int num_copies;
        ChInterp_t interp;
        ChVaList_t ap;
        int ret;

        Ch_VaStart(interp, ap, varg);
        agent = Ch_VaArg(interp, ap, MCAgent_t);
        num_copies = Ch_VaArg(interp, ap, int);
        ret = MC_SetAgentToCopy(agent, num_copies);
        Ch_VaEnd(interp, ap);

        return ret;
}
```

MC_SetDestinationsToCopy() i *MC_SetDestinationsToCopy_chdl()*

MC_SetDestinationsToCopy() s'encarrega d'afegir a l'estat temporal, *datastate*, de cada agent quines destinacions volem que tinguin, cas que es copiï, les seves còpies. Cada vegada que cridem a la funció afegirem un nou destí a la llista. Cas que sigui el primer destí que afegim a un cert *datastate* haurem de reservar memòria per a l'*array* de cadenes de caràcters que contindran els destins i el destí que vulguem afegir. Si no és així només ens caldrà reservar memòria pel destí que vulguem afegir. *agent* és l'agent que es vol copiar, i *destination* una cadena de caràcters amb el destí a afegir. La funció retorna MC_SUCCESS cas que s'hagi executat sense errors.

Pel que fa a *MC_SetDestinationsToCopy_chdl()* és una funció que serveix perquè *MC_SetDestinationsToCopy()* es pugui cridar des de l'espai d'*script*.

A continuació reproduïm el codi de *MC_SetDestinationsToCopy()*:

```
EXPORTMC int MC_SetDestinationsToCopy(MCAgent_t agent, char*
    destination)
{

        MUTEX_LOCK(agent->lock);

        if (agent->datastate->num_copies != -1){    // Don't copy the
            agent more than a time.

                if (agent->datastate->destination_copies == NULL){
                  agent->datastate->destination_copies = (char**)
                      malloc (sizeof(char*)*MAX_DESTINATIONS);
                }
```

```
                agent->datastate->destination_copies[agent->datastate
                    ->num_destination_copies] = (char*) malloc (
                    sizeof(char)*MAX_DESTINATION);
                strcpy(agent->datastate->destination_copies[agent->
                    datastate->num_destination_copies], destination);
                agent->datastate->num_destination_copies++;

        }

        MUTEX_UNLOCK(agent->lock);

        return MC_SUCCESS;
}
```

MC_SetLsToCopy() i *MC_SetLsToCopy_chdl()*

Anàlogament, la funció *MC_SetLsToCopy()* s'encarrega d'afegir a l'estat temporal, *datastate*, de cada agent el paràmetre L que li correspongui segons haurà explicitat el codi d'encaminament que haurà afegit una sèrie de paràmetres L a la llista segons la implementació d'*spray and wait* d'aquest projecte.

Pel que fa a *MC_SetLsToCopy_chdl()* és una funció que serveix perquè *MC_SetLsToCopy()* es pugui cridar des de l'espai d'*script*.

A continuació reproduïm el codi de *MC_SetLsToCopy()*:

```
EXPORTMC int MC_SetLsToCopy(MCAgent_t agent, int L)
{

        MUTEX_LOCK(agent->lock);

        if (agent->datastate->num_copies != -1){    // Don't copy the
            agent more than a time.

                if (agent->datastate->L_copies == NULL){
                  agent->datastate->L_copies = (int*) malloc (sizeof(
                      int)*MAX_DESTINATIONS);
                }

                agent->datastate->L_copies[agent->datastate->
                    num_L_copies] = L;
                agent->datastate->num_L_copies++;

        }

        MUTEX_UNLOCK(agent->lock);

        return MC_SUCCESS;
}
```

MC_GetNumberOfCopies() i *MC_GetNumberOfCopies_chdl()*

MC_GetNumberOfCopies() retorna el nombre de còpies a fer d'un agent en un moment determinat. *agent* és l'agent que es vol copiar del qual volem saber el nombre de còpies.

Pel que fa a *MC_GetNumberOfCopies_chdl()* és una funció que serveix perquè *MC_GetNumberOfCopies()* es pugui cridar des de l'espai d'*script*.

La definició de *MC_GetNumberOfCopies()* és la següent:

```
EXPORTMC int MC_GetNumberOfCopies(MCAgent_t agent)
{
        MUTEX_LOCK(agent->lock);
        int copies = agent->datastate->num_copies;
        MUTEX_UNLOCK(agent->lock);
        return copies;
}
```

MC_GetCongestion() i *MC_GetCongestion_chdl()*

MC_GetCongestion() retorna la congestió de la plataforma que no és res més que la mida de la seva cua d'agents la qual ens permet saber en tot moment quina quantitat de dades estan pendents d'enviar-se a la plataforma.

Pel que fa a *MC_GetCongestion_chdl()* és una funció que serveix perquè *MC_GetCongestion()* es pugui cridar des de l'espai d'*script*.

```
return g_mc_platform->agent_queue->size;
```

MC_Get/SetLParameter() i MC_Get/SetLParameter_chdl()

MC_SetLParameter() s'encarrega d'afegir a l'agent el paràmetre *L*. *agent* és l'agent que es vol copiar, i *L* el paràmtre *L* d'*spray and wait*. La funció retorna MC_SUCCESS cas que s'hagi executat sense errors.

Pel que fa a *MC_SetLParameter_chdl()* és una funció que serveix perquè *MC_SetLParameter()* es pugui cridar des de l'espai d'*script*.

MC_GetLParameter() s'encarrega de retornar el paràmetre L que hem afegit amb amb *MC_SetLParameter()*. El paràmetre *agent* és l'agent que es vol copiar. La funció retorna el valor del paràmetre L cas que s'hagi executat sense errors.

Pel que fa a *MC_GetLParameter_chdl()* és una funció que serveix perquè *MC_GetLParameter()* es pugui cridar des de l'espai d'*script*.

El codi de *MC_SetLParameter_chdl()* i *MC_GetLParameter_chdl()* és el següent:

```
EXPORTMC int MC_SetLParameter(MCAgent_t agent, int L)
{
        MUTEX_LOCK(agent->lock);
        agent->L = L;
        MUTEX_UNLOCK(agent->lock);
        return MC_SUCCESS;
}
```

```
EXPORTMC int MC_GetLParameter(MCAgent_t agent)
{
        return agent->L;
}
```

libmc.h

A *libmc.h* hi ha la declaració de les funcions definides a *libmc.c* Hem afegit la paraula reservada *extern* tot i que no és necessària. En C, s'afegeix implícitament a les declaracions de funcions però l'hem afageit per mantenir el mateix estil de codi que l'existent.

Codi de la Definició de l'Agent

Aquí farem cinc cèntims dels canvis al codi que defineix l'estructura de l'agent.

agent.c

agent.c conté una sèrie de funcions per manipular agents, copiar-los, inicialitzar-los...

Nosaltres hem modificat la funció que s'utilitza per copiar (*agent_Copy()*) agents per poder fer-la servir per implementar l'*spray and wait* ja que la implementació que hi havia ocasionava errors en algunes situacions. Els canvis que hem fet són els següents:

A la funció *agent_Copy()* hem posat tota la memòria que ocuparà la còpia de l'agent a 0s amb *memset*.

```
memset(cp_agent, 0, sizeof(agent_t));
```

A la mateixa funció *agent_Copy()* copiem el flag que ens indica si un agent és binari. Si no ho fem quan es copia algun agent no binari podia donar-se el cas que la còpia no fos binària o a l'inrevés depenent del que hi hagués anteriorment en la posició de memòria corresponent al flag.

```
cp_agent->binary = agent->binary;
```

En el codi original, a *agent_Copy()* no es copia el camp sender. Ho hem afegit amb següent codi:

```
if (agent->sender != NULL){
    cp_agent->sender = (char*)malloc
      (
       sizeof(char) *
       (strlen(agent->sender) + 1)
       );
    strcpy(cp_agent->sender, agent->sender);
}
else{
    cp_agent->sender = NULL;
}
```

Finalment, desbloquegem l'agent que hem copiat puix que en el codi original es quedava bloquejat de forma indefinida.

```
MUTEX_UNLOCK(agent->lock);
```

A *agent_ChScriptInitVar()* afegim les noves variables de l'espai d'*script* de Ch. Sense això no seria possible cridar les funcions que s'han definit a *libmc.c* des del codi d'encaminament perquè Ch, que és l'entorn d'execució utilitzat, no les reconeixeria.

```
Ch_DeclareFunc(
    *interp,
    "int MC_GetNumberOfCopies(MCAgent_t agent);",
    (ChFuncdl_t)MC_GetNumberOfCopies_chdl
    );
Ch_DeclareFunc(
    *interp,
    "int MC_GetCongestion();",
    (ChFuncdl_t)MC_GetCongestion_chdl
    );
Ch_DeclareFunc(
    *interp,
    "int MC_SetLParameter(MCAgent_t agent, int L);",
    (ChFuncdl_t)MC_SetLParameter_chdl
    );
Ch_DeclareFunc(
    *interp,
    "int MC_GetLParameter(MCAgent_t agent);",
    (ChFuncdl_t)MC_GetLParameter_chdl
    );
```

agent_datastate.c

Aquest arxiu conté totes les funcions per manipular el *datastate*, copiar-lo, inicialitzar-lo...

El *datastate* és el conjunt de dades que configura l'estat temporal d'un agent com, per exemple, el pròxim node a saltar.

En aquest projecte hem afegit una sèrie de variables al *datastate* per guardar les destinacions a què s'ha de copiar un agent i les *Ls* de les còpies. A més a més, hem hagut de modificar algunes funcions que no funcionaven satisfactòriament en alguns casos.

A la funció *agent_datastate_Copy()* afegim els nous camps del datastate que hem afegit. Pel que fa a *num_copies* no copiarem el contingut del camp de l'agent original al de la còpia sinó que al camp copiat li posarem un -1 per indicar que no volem copiar un altre cop un agent que ja hem replicat. Per la mateixa raó posarem a NULL les destinacions on enviar les còpies i a 0 el número d'aquestes.

```
cp_data->is_routable = datastate->is_routable;
strcpy(cp_data->destination, datastate->destination);

cp_data->num_copies = -1;
cp_data->destination_copies = NULL;
cp_data->num_destination_copies = 0;
```

També a *agent_datastate_Copy()* posem a NULL el codi d'encaminament i les *ids* d'aquest codi d'encaminament perquè, en tot cas, aquest codi s'afegirà després.

```
cp_data->routing_code_ids[num_routing_code_ids] = NULL; // JOSEP
cp_data->routing_codes[num_routing_code_ids] = NULL; // JOSEP
```

Per últim hem corregit un error també a *agent_datastate_Copy()* en que es començava un bucle amb un índex que no tenia per què ser 0 tal com estava concebut. Per això inicialitzem a 0 la variable *i* abans del bucle *while* que és codi ja existent.

```
/* Look for a routing code with that id */
i = 0; // JOSEP
while (i < num_routing_code_ids)
{
        if (strcmp(buf, cp_data->routing_code_ids[i]) == 0)
        {
                cp_data->routing_code = cp_data->routing_codes[i];
        }
        i++;
}
```

A *agent_datastate_New()* afegim la inicialització dels camps nous del datastate:

```
agent_datastate->num_copies = 0;
agent_datastate->destination_copies = NULL;
agent_datastate->num_destination_copies = 0;
```

agent_task.c

agent_task.c s'encarrega de les tasques que fa un agent. A diferència del cas anterior no hi hem afegit cap variable però també hi haurem de fer algunes modificacions perquè s'executi satisfactòriament.

A la funció *agent_task_Copy()* hem afegit la següent línia:

```
cp_task->agent_return_data = NULL;
```

En aquesta funció, per a copiar tasques, no es tenia en compte que el camp *agent_return_data* de les tasques de les còpies dels agents no podia apuntar a qualsevol lloc. Això provocava errors de segmentació en alguns casos.

També a la funció *agent_task_Copy()* hem afegit el següent *if*:

```
if (task->routing_code_id != NULL){
cp_task->routing_code_id = strdup(task->routing_code_id);
```

Això fa que cas que quan *routing_code_id* sigui NULL no es copiï com si fos una cadena de caràcters cosa que comportaria una errada de segmentació.

4.2 El mòdul de descobriment de veïns: *prosesd*

A *prosesd* simplement ens hem assegurat que quan es rebi un missatge d'algun veí on s'inclogui la congestió, aquesta s'afegeixi a l'ontologia que el *prosesd* enviarà a la plataforma del node on estigui.

Com a ontologia considerem una sèrie de dades sobre la plataforma que passem a l'agent.

Això es fa afegint el següent a *net_monitor.c*:

```
char onto_congestion[40];
snprintf(onto_congestion, sizeof(onto_congestion),"CONGESTION:%d",
    packet.congestion);
ontodata_property(onto_congestion,SET);
```

A *net_monitor.h* haurem de modificar l'estructura de les sol·licituds, *cmd_hdr*, perquè s'ha de tenir en compte que s'enviarà un camp més, la congestió.

El codi modificat és el següent:

```
typedef struct
{
  unsigned short type;
  unsigned short timeout;
  unsigned int congestion;
} cmd_hdr;
```

Si no ho féssim no hi hauria cap error d'execució perquè a l'origen s'enviaria l'estructura correcta, ara bé el *prosesd* no podria prendre el valor de la congestió i això provocaria resultats inesperats.

4.3 Codi d'Encaminament

Per afegir el codi d'encaminament als agents ens hem valgut d'una modificació d'un dels exemples que el dEIC va introduir en el codi de Mobile-C. Aquest és *MCPACKAGE/demos/routing_code/jumping_routing*.

El resultat és el següent:

```
int main ()
{
        int congestion_level = 5; // congestion level
        int L; // spray and wait number of copies
        int Laux;

        printf("Executing Routing Code...\n");

        sleep(2);

        /* Receive the information about actual neighbours*/
        // Pointer to the struct that stores the information
        struct mc_neighbour *p;
        int i,j=0;
        int total_copies = 0;
        int num_neighbours = 0;
        char *new_destination = NULL;
        char *port = (char*) malloc (sizeof(char)*MAX_PORT+1);
        strcpy(port, "5052");


        if (mc_host_port == 5050){ // client

                mc_SetAgentNextHop(mc_current_agent, "localhost:5052"
                    );

        } else{ // server

                // input variables
                int alpha = 20;
                int qo = MC_GetCongestion() + 1;

                // output variables
                int L;

                p = mc_GetNeighbours();
                num_neighbours = mc_GetNumOfNeighbours();

                printf("num_neighbours: %d\n", num_neighbours);
```

```
if (!(strcmp(mc_host_name,"miniubuntu-aux5")) &&
    MC_GetNumberOfCopies(mc_current_agent) != -1 &&
    num_neighbours > 0){ // Estem al node origen.

        // calculus of L

        L = ROUND((float) alpha / (float) qo);
        L++;
        printf("L: %d\n", L);

}
else{   // No estem al node origen.
        L = MC_GetLParameter(mc_current_agent);
}

if (!(strcmp(mc_host_name,"miniubuntu-aux5")) &&
    num_neighbours > 0 && MC_GetNumberOfCopies(
    mc_current_agent) != -1){


        int *congestions = (int*) malloc (sizeof(int)
            * num_neighbours);
        int *Ls = (int*) malloc (sizeof(int) *
            num_neighbours);
        int aux = 0;

        // initialize congestions
        congestions[0] = qo;
        for (i=1; i<num_neighbours+1; i++){
          congestions[i] = 1 + getCongestion(p[i-1]);
        }

        Ls = lcmAndNumeratorSum(congestions,
            num_neighbours+1, L);

        MC_SetLParameter(mc_current_agent, Ls[0]);

        // debug
        printf("L: %d %s\n", L, mc_agent_name);
        printf("congestions: %d %d %d\n", congestions
            [0], congestions[1], congestions[2]);
        printf("Ls: %d %d %d\n", Ls[0], Ls[1], Ls[2])
            ;

        for (i=0; i<num_neighbours; i++){
          if (Ls[i+1] > 0){
            new_destination = (char*) realloc (
                new_destination, sizeof(char)*(strlen
                (p[i].ip)+strlen(port)+1)+1);
            strcpy(new_destination, p[i].ip);
            strcat(new_destination, ":");
            strcat(new_destination, port);
            MC_SetDestinationsToCopy(mc_current_agent
                , new_destination);
            MC_SetLsToCopy(mc_current_agent, Ls[i+1])
                ;
          }
          else{
```

```
                                aux++;
                            }
                        }
                        free(new_destination);
                        MC_SetAgentToCopy(mc_current_agent,
                            num_neighbours-aux);

                    }
                }

    return 0;
}
```

En aquest codi podem observar com actuarem de manera diferent depenent de si ens trobem en el client o en el servidor.

El client només és una eina per enviar-li agents al servidor. Per tant, si ens trobem al client simplement li indicarem a l'agent que el seu proper node serà el del servidor (en el nostre cas utilitzem un port diferent de la mateixa màquina virtual per representar-lo).

En canvi, si ens trobem al servidor haurem de calcular, en primer lloc, el paràmetre L d'*spray and wait*.

El càlcul cas que estiguem a l'origen es fa de la següent manera. No és possible determinar un valor de L òptim sense saber el nombre de nodes de la xarxa per tant haurem de buscar una heurística que sigui suficientment satisfactòria i que no necessiti el nombre de nodes a la DTN.

Podem considerar la metonímia que com més congestionat estigui l'origen més congestionada estarà la xarxa. Com més congestionada estigui la xarxa menys agents enviarem per evitar congestionar-la més i a l'inrevés. Per tant fem que L sigui inversament proporcional a la congestió del node origen q_0. Afegirem un *1* a la fórmula perquè no volem que en cap cas L inicial sigui 0 (voldrem enviar, com a mínim, una còpia del missatge).

$L = ROUND(alpha/qo) + 1;$

.. on *alpha* és un factor de proporcionalitat i q_o és la congestió de l'origen.

Aquesta L s'ha de repartir entre els veïns de manera inversament proporcional a la congestió de cada veí. Cal remarcar que tot i que en un xarxa real això es farà a tots els nodes aquí només es farà a l'origen (*miniubuntu-aux5*) perquè puix que en la nostra xarxa els nodes es veuen els uns als altres en tot moment si no hi hagués aquesta condició els agents es tornarien a enviar a l'origen i un altre cop als seus veïns indefinidament.

Per il·lustrar aquest procés ens valdrem de les següents variables:

q_i: Congestió del veí i més una unitat per evitar tenir congestió 0 que ocasionaria problemes amb els inversos.

q_t: Total d'agents a enviar per tots els veïns calculat a partir del conjunt de q_is.

q_m: mitjana d'agents dels veïns calculada a partir del conjunt de q_is.

num_neighbours: nombre de veïns

Primera aproximació:

$L_i = \frac{q_m}{q_i} L$

No funciona perquè el sumatori de les L_i no dóna L. La solució correcta és:

Passem $\frac{q_m}{q_i}$ per i $\in (0..num\_neighbours)$ on n és el nombre de veïns a comú denominador i anomenem n_is els numeradors per i $\in (0..num\_neighbours)$.

Agafem la suma dels numeradors, β, i tenim:

$L_i = \frac{q_t n_i}{\beta}$ per i $\in (0..n)$ on n és el nombre de veins, a comú denominador.

Això ho fem mitjançant la funció *lcmAndNumeratorSum()*. S'ha de tenir en compte que la funció descrita, a més a més de repartir la L de manera inversament proporcional a la congestió, s'encarrega d'arrodonir els resultats a enters i d'assegurar-se que segueixen sumant L. Com a conseqüència d'això per una part d'aquesta funció hem utilitzat l'algorisme de selecció *select sort* prenent com a mostra la definició de [1].

A continuació, s'indiquen a la plataforma les destinacions a les quals voldrem enviar els agents i les *Ls* de cada còpia que acabem de calcular. Finalment s'indica a la plataforma el número de còpies que es vol fer de l'agent (Això és una per cada veí amb una L més gran que 0).

4.4 Discussió sobre la Implementació d'aquest PFC

En aquesta secció discutirem breument per què la implementació en l'entorn utilitzat és més costosa pel que fa al temps que en la majoria de situacions.

Moltes eines de depuració de codi com ara *dbg* o *Valgrind* no són d'utilitat en entorns com ara el de Mobile-C ja que el fet que es treballi amb C

interpretat mitjançant *Ch* impedeix en molts casos l'execució d'aquestes eines.

Com que en la majoria de casos no hem pogut utilitzar els instruments tradicionals ens hem vist obligats a utilitzar mètodes molt més tradicionals. Bàsicament hem imprès per pantalla el valor de variables, o simplement missatges per tal de localitzat els errors de segmentació. En el *prosesd*, a més a més, hem hagut de cridar-lo amb el *flag -d* per indicar que volem que els missatges s'imprimeixin en pantalla i no només als *logs*.

Aquest mètode de depuració de codi ha estat feixuc perquè era necessari re-compilar tot el codi cada vegada que es volia afegir una nova instrucció d'impressió per pantalla. Això pot tardar uns quants minuts quan treballem amb un codi tan gran com Mobile-C.

A més a més, el fet que Mobile-C està basat en *threads* complica molt més aquesta tasca ja que en molts moments s'estan executat fins a cinc *threads* de forma paral·lela en cada màquina virtual i depurar errors en aquestes condicions requereix una gran quantitat de temps, i paciència.

Un punt també conflictiu d'aquest projecte ha estat la instal·lació de tot l'entorn de Mobile-C en màquines virtuals. Ens han donat problemes fets com ara que si s'instal·la Mobile-C i després *prosesd* el codi de Mobile-C no s'executa correctament però sí si es fa en ordre invers.

És més, tot i que hi ha manuals d'instal·lació de Mobile-C a la pàgina web del projecte hem trobat a faltar un manual on s'expliqués la instal·lació des de zero en una màquina virtual incloent-hi les modificacions per treballar amb DTNs del dEIC (i ara també la implementació de l'*spray and wait*). Per aquesta raó ens hem permès la llibertat d'explicar breument tots els passos i l'ordre que hem seguit per instal·lar-ho en una màquina virtual perquè tothom, independentment de l'entorn amb el qual treballi, ho pugui fer. Hem inclòs aquesta explicació a l'annex.

Capítol 5

Proves

En aquest capítol descriurem breument les proves que s'han realitzat per evaluar la correctesa dels canvis efectuats.

L'objecte d'aquest projecte ha estat la implementació d'un mecanisme d'encaminament concret i no la generació d'escenaris per realitzar-hi proves puix que això podria constituir un projecte de final de carrera per si mateix.

Tot i això ens hem vist a realitzar algunes proves per demostrar la validesa de l'algorisme implementat.

Mitjançant scripts en *bash* i un grup de tres màquines virtuals (VirtualBox) amb Ubuntu 12.04 que actuen com a nodes diferents hem simulat congestió en algunes màquines activant el *prosesd* de forma intermitent i enviant un nombre elevat d'agents.

D'aquesta manera hem pogut observar que l'origen envia un nombre més gran de missatges als veïns menys congestionats i n'envia més als que tenen un nivell de congestió més elevat.

Això concorda amb l'objectiu d'aquest treball.

Més concretament, l'esquema utilitzat és el que es mostra a la figura 5.1.

Tenim un node origen en el qual un client envia un agent a un servidor el qual calcula el paràmetre L d'*spray and wait* i reparteix L còpies entre els seus veïns i ell mateix de forma inversament proporcional a la congestió de cada node.

A continuació mostrem una sèrie de proves concretes que s'han fet.

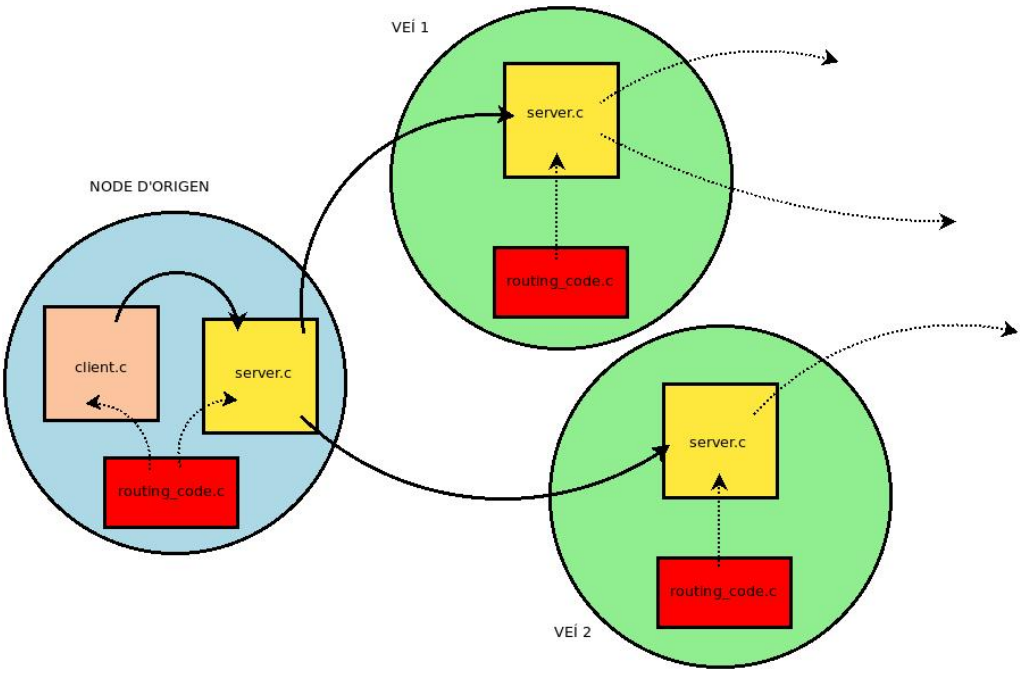

Figura 5.1: Esquema general. Tenim tres màquines virtuals que representen tres nodes d'una xarxa.

A la figura 5.2 es mostra com el node origen té una L que val 11. Això vol dir que la congestió és baixa (concretament val *2*) i, per tant, ens podem permetre enviar un gran nombre d'agents a la xarxa. El nivell de congestió és correcte perquè en aquest cas només havíem enviat un agent des del client al servidor del node origen i sempre se li suma una unitat a la congestió perquè cas que la cua d'agents fos 0 tindriem congestió 0 i això ens donaria problemes en treballar amb inversos. Com que els seus veïns tenen un nivell de congestió menys elevat (ambdós tenen nivell 1, és a dir cap agent a la cua) els tocaran més còpies (cinc a un i quatre a l'altre) que no pas a l'origen (dues).

Observem que es compleix la propietat que L equival al nombre de còpies repartides 2+5+4=11.

A la figura 5.3 es mostra com el node origen té també una congestió de 3. Com que la congestió és major que en el cas anterior L serà menor. Efectivament, val 8.

Pel que fa a la manera de distribuir les còpies, es reparteixen de la manera més equitativa possible entre els tres nodes. Una còpia pel primer node,

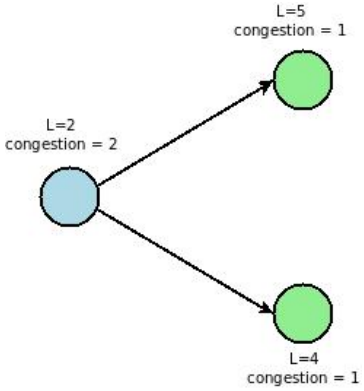

Figura 5.2: Exemple d'spray and wait amb L inicial 11

quatre pel segon i tres pel tercer. Aquest repartiment s'hauria pogut fer de manera diferent (donar-li 3 còpies a l'origen i a un veí i 4 a l'altre però com que no hi ha cap manera millor que l'altra l'algorisme simplement en decideix una).

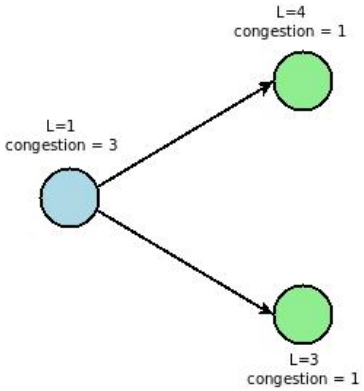

Figura 5.3: Exemple d'spray and wait amb L inicial 8

A la figura 5.3 es mostra com el node origen té també una congestió de 6. Com que la congestió és encara major que en el cas anterior L serà menor. Efectivament, val 4.

Pel que fa a la manera de distribuir les còpies, l'origen no es queda cap còpia i reparteix les quatre còpies de manera equitativa entre els dos veïns.

Finalment a la figura 5.5 es mostra com repartir 6 còpies entre, aquest cop, cinc nodes. Tenim un nou escenari amb 5 màquines virtuals contant l'origen amb diferents congestions.

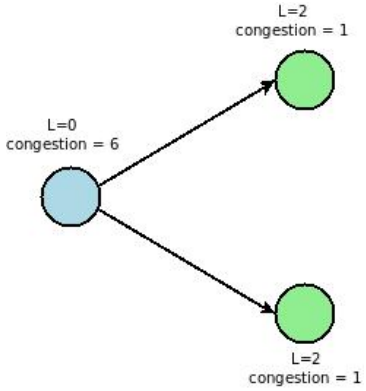

Figura 5.4: Exemple d'spray and wait amb L inicial 4

A l'hora de distribuir les còpies, l'origen es queda la majoria de còpies (3) perquè és el node més congestionat (4) i n'envia una als tres veïns menys congestionats. L'altre veí, que té una gran congestió (224) no es queda cap còpia.

Com en tots els casos es pot veure com la suma de les còpies (3+1+1+0+1=6) és la L inicial ja que L és el nombre de còpies que hi haurà a la xarxa mentre no s'arribi al destí.

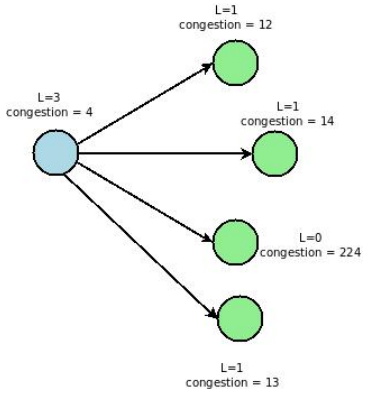

Figura 5.5: Exemple d'spray and wait amb L inicial 6

Capítol 6

Conclusions

La conclusió principal d'aquest projecte ha estat que és possible, mitjançant la modificació d'una plataforma d'agents mòbils com ara Mobile-C, implementar un mecanisme d'encaminament complex com ara *spray and wait*.

Ara bé, també hem comprovat que no és immediat perquè hi ha multitud de restriccions, principalment el fet que un agent no es pot clonar directament des del seu codi d'encaminament. Per fer això hem hagut de delegar-ho a la plataforma a la qual pertanyen els agents. Aquesta plataforma, còpia els agents quan així se li indica des del codi d'encaminament amb uns determinats paràmetres (destinació i L) que també se li donen des del codi d'encaminament que és qui té la capacitat de calcular-los.

D'altra banda cal destacar que tot i que la correció del codi dels projectes de codi obert com ara Mobile-C és elevada és freqüent trobar-hi errades ja que tant l'extensió com la complexitat del codi fan impossible que estigui lliure d'errors.

Això és un factor que s'ha de tenir sempre en compte, especialment quan volem utilitzar un codi com aquest d'una manera lleugerament diferent a l'utilitat per la qual s'ha concebut.

En el cas de Mobile-C funcions com ara la de copiar un agent ocasionaven errors en executar-les. La majoria de vegades perquè s'alliberava memòria doblement. La primera vegada que s'alliberava un certa zona de memòria la instrucció *free* s'executava correctament però la segona vegada ocasionava una errada de segmentació puix que aquella zona de memòria que s'intentava alliberar ja no estava reservada.

També és necessari tenir en compte aquest factor en la planificació temporal del projecte. La correcció d'errors en codis com el de Mobile-C comporta un espai de temps elevat que no sempre es té en compte.

Com hem exposat al capítol sobre la implementació, en entorns com l'utilitzat, la depuració i, per tant, el desenvolupament de codi és molt més costós que en la majoria de situacions perquè no es poden utilitzar les eines de depuració habituals i, a més a més, es tracta d'un codi amb un alt nivell de paral·lelització amb diferents fils d'execució.

D'altra banda, el fet d'haver treballat en un entorn de recerca com és el dEIC, ens ha proporcionat *in situ*, l'experiència de realitzar, tot i que en un nivell elemental, en certa manera, recerca. Això ha estat una experiència molt positiva. A més a més, treballar amb Mobile-C, ha estat de gran profit perquè ens ha permès treballar amb un codi complex i de qualitat cosa que és molt enriquidora.

Més concretament, podem dir que s'han complert cada un dels objectius enumerats al principi d'aquesta memòria.

- Estudiar els principals mètodes d'encaminament de DTNs: S'han estudiat els principals mètodes d'encaminament de les DTNs per tenir una base a partir de la qual desenvolupar el projecte.
- Comprendre i modificar la plataforma Mobile-C per realitzar un encaminament més eficaç: Això s'ha assolit amb escreix perquè per solucionar les mancances del codi original i preparar-lo per poder implementar l'*spray and wait* ens hem vist obligats a conéixer a fons el funcionament de la plataforma.
- Comprendre i modificar el dimoni *prosesd* per intercanviar informació entre plataformes: Això també s'ha assolit ja que, de fet, s'ha hagut de modificar el *prosesd* perquè els agents es podessin intercanviar la congestió entre ells.
- Estudiar i implementar el mecanisme d'*spray and wait* utilitzant l'esquema d'optimització a l'origen: Assolit tal com demostren les proves de l'apartat anterior.

6.1 Revisió de la Planificació

Sobre la distribució del temps, cal dir que al final només hem implementat un dels quatre mecanismes d'*spray and wait*, l'optimització a l'origen. La

principal raó per la qual hem implementat només un mecanisme és que hem invertit molt de temps en modificar la plataforma Mobile-C perquè un agent es poguéssin copiar agents metre s'està executant i en corregir errors del codi original. Això és una activitat amb què no havíem comptat i que afecta visiblement el diagrama de Gantt final. Pel que fa a la redacció de la memòria, aquesta ha estat una activitat que s'ha anat desenvolupat durant bona part de la realització d'aquest projecte tot i que de manera indirecta ja que s'ha anat editant una *wiki* que posteriorment s'ha traslladat en bona part a aquest document.

El diagrama de Gantt de la planificació final es mostra a la figura 6.1.

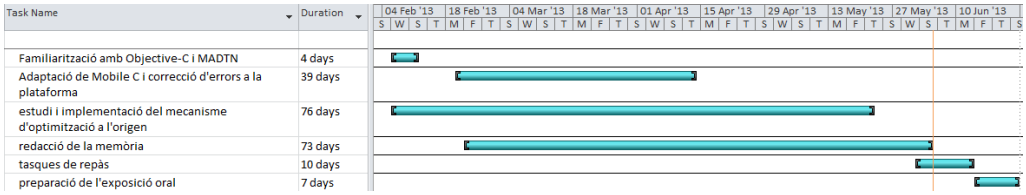

Figura 6.1: Diagrama de Gantt de la planificació temporal real

6.2 Treball Futur

Pel que respecta al treball futur, un següent pas lògic seria implementar els altres tres esquemes d'*spray and wait*, *optimització al node*, *control de switching* i *gain scheduling control* ja que amb les modificacions realitzades a Mobile-C i *prosesd* no suposaria una càrrega de feina excessivament gran.

També seria interessant afegir una nova heurística per calcular la L i repartir-la que tingui en compte més factors a més a més de la congestió.

D'altra banda, i això entraria dins un àmbit diferent, seria molt interessant que algú que en el futur que realitzés un projecte de generació d'escenaris aleatoris de simulació comprovés fins a quin punt aquest esquemes milloren l'*spray and wait* per poder-los modificar de forma paral·lela i determinar una estratègia que funcioni satisfactòriament per a tots els casos.

Bibliografia

[1] *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, 2007.

[2] Akan O. B. Chen C. Su-W. Akyildiz, I. F. *InterPlaNetary Internet: state-of-the-art and research challenges*, 2003.

[3] C. Borrego. *A Mobile Code-based Multi-Routing Protocol Architecture for Delay and Disruption Tolerant Networking*, 2013.

[4] Gallagher B. Jensen D. Neil Levine B. Burgess, J. *MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks*, 2006.

[5] WageIndicator Foundation. Tusalario.es, 2013.

[6] Oki H. Wang Y. Margaret Martonosi Li-Shiuan Peh Juang, P. and Daniel Rubenstein. *Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet*, 2002.

[7] Assi C. M. Fawaz W. F. Khabbaz, M. J. *Disruption-Tolerant Networking: A Comprehensive Survey on Recent Developments and Persisting Challenges*, 2012.

[8] Doria A. Lindgren, A. *Probabilistic Routing Protocol for Intermittently Connected Networks draft-irtf-dtnrg-prophet-09*, 2009.

[9] Robles S. Martí R. Sreenan C. J.-Borrell J. Mercadal, E. *Heterogeneous Multiagent Architecture for Dynamic Triage of Victims in Emergency Scenarios*, 2011.

[10] E. Nordström. Haggle faq, 2011.

[11] Chai S. Huang Z. Pavlovich, N. L. *Opportunistic Medical Data Delivery in Challenged Environments*, 2010.

[12] Nadjm-Tehrani S. Sandulescu, G. *Optimising Replication versus Redundancy in Window-aware Opportunistic Routing*, 2010.

[13] Psounis-K. Raghavendra C. Spyropoulos, T. *Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks*, 2005.

[14] Chatterjee-M. Kwiat K. Venkataraman, M. *A dynamic reconfigurable routing framework for wireless sensor networks*, 2011.

ANNEX: Posada a Punt de Mobile-C

Hem utilitzat una màquina virtual (VirtualBox) perquè la instal·lació del Mobile-C a la nostra màquina de 64 bits donava problemes en compilar les llibreries necessàries.

Per instal·lar a la màquina virtual hem escollit un sistema operatiu que fos lleuger però a la vegada potent. Hem escollit Ubuntu Mini de 32 bits, una versió d'Ubuntu que permet instal·lar exclusivament el software que es necessitarà i evitar, així, tenir un sistema operatiu que consumeixi massa recursos en una màquina virtual. Durant la instal·lació hem especificat que volem instal·lar LVM per poder tenir carpetes compartides amb la màquina real de manera més àgil.

Tenint en compte que en el transcurs d'aquest treball serà necessari treballar amb més d'un terminal obert hem instal·lat un sistema de finestres; el *Xorg*. També necessitem un escriptori lleuger. Hem escollit el FVWM.

També hem hagut d'instal·lar un compilador de C, *g++*, necessari per compilar Mobile-C i també autoconf necessari per instal·lar Mobile-C.

Per editar el codi necessitem un programa funcional però amb poc consum de recursos. Hem escollit l'*Emacs*.

També instal·lem el *bc*, un programa necessari per realitzar càlculs de punt flotant en bash.

Per últim hem instal·lat el dimoni de descobriment de veïns prosesd del dEIC, necessària per certes funcions del Mobile-C i la llibreria *libgps-dev* necessària per instal·lar *prosesd*.