Smart Contracts sobre Bitcoin

MISTIC: Màster Interuniversitari en Seguretat de les Tecnologies de la Informació i de les Comunicacions

Treball de fi de màster dels estudis del Màster Interuniversitari en Seguretat de les Tecnologies de la Informació i de les Comunicacions presentat per Josep Miquel Andreu i dirigit per Jordi Herrera.

Universitat Oberta de Catalunya, Barcelona, 2016

Resum

El present treball final de màster realitza una introducció als *Smart Contracts*. El treball introdueix el concepte de contracte intel·ligent, els seus usos i alguns exemples existents. Seguidament proporciona les nocions necessàries de les transaccions del protocol Bitcoin per a poder implementar un contracte intel·ligent, usant la blockchain que ofereix el protocol. Per últim, s'explica la implementació d'un contracte intel·ligent usant bitcoin: un canal de micropagaments.

A l'Àlex, al Pau, al Joan i a la Laura

Agraïments

A la meva família, per tots els esforços que han fet per a que jo pugui estar escrivint aquest projecte.

A l'Àlex, per ajudar-me en els moments més difícils.

Al Pau, al Joan i a la Laura per conviure amb mi mentre ha durat aquesta aventura.

I al Jordi Herrera, per guiar-me en el fascinant i alhora complexe món dels bitcoins.

Índex

Índex de figures

Índex de taules

CAPÍTOL 1

Introducció

El Bitcoin probablement és la moneda virtual descentralitzada més coneguda actualment. Al no dependre de cap entitat centralitzada, obre moltes possibilitats de nous usos. Una funcionalitat no tan coneguda és l'oportunitat d'usar la seva infraestructura per crear smarts contracts.

Un smart contract és un contracte que és capaç d'executar-se o fer-se complir a ell mateix. Habitualment, en l'execució d'un contracte és necessària la implicació de tercers. Un smart contract ens permet ometre aquests tercers.

Els smart contracts s'escriuen en llenguatges de programació que permeten definir regles i conseqüències estrictes segons el compliment d'aquestes regles. Això obre la porta a molts dubtes: on s'executen aquests contractes, com obtenen la informació necessària per a executar les conseqüències, quin tipus de conseqüències es poden implementar, si es poden modificar...

En aquest projecte intentarem donar resposta a totes aquestes preguntes per a després implementar un contracte usant la infraestructura Bitcoin. Per fer-ho, usarem la testnet de Bitcoin i el llenguatge scripting que ofereix la moneda.

1.1 Objectius

La línia de treball principal del projecte és analitzar els smart contracts i implementar-ne un mitjançant les eines que ofereix el Bitcoin. Per poder implementar el contracte, serà necessari assolir uns coneixements previs sobre el funcionament dels smart contracts, la xarxa test de Bitcoin (on implementarem el contracte) i el llenguatge scripting que ofereix Bitcoin.

Per concretar la línia de treball, es plantegen els següents objectius:

1. Estudiar què és un smart contract, els seus usos, tipus i funcionaments.

2. Descriure com usar la testnet de Bitcoin per poder implementar un smart contract.
3. Estudiar el llenguatge scripting que ofereix Bitcoin per a poder ser usat en la implementació d'un smart contract.
4. Implementar un smart contract.

El treball no té com a objectiu entendre el funcionament del Bitcoin. Aleshores, per a la seva comprensió es pressuposa que el lector té uns coneixements mínims del funcionament de la moneda.

1.2 Requeriments, recursos i viabilitat

El projecte conté una fase important de recerca i anàlisi on no hi ha un consum específic de requeriments o recursos. Pel que fa la implementació del contracte no són necessaris recursos amb elevats costos. Únicament és necessari un ordinador amb accés a Internet i que pugui accedir a la testnet de Bitcoin.

Amb el temps que es disposa i l'absència de realitzar inversions, el projecte és viable de portar a terme.

1.3 Planificació temporal

El projecte està pensat per realitzar-se durant tot el semestre Setembre - Febrer. Aquest està dividit en dos grans blocs: anàlisi i implementació. Les dates d'entrega de les diferents PAC marquen el ritme del projecte.

Pel que fa la fase d'anàlisi, està previst que finalitzi poc abans de l'entrega de la segona PAC. Està subdividida en dos blocs. Per una banda, la recerca d'informació dels smart contracts i per altra la informació necessària per a usar la testnet de Bitcoin. En el primer bloc s'inclourà el funcionament d'un smart contract, usos, exemples i tipus. La segona part es centrarà en com connectar-se a la testnet i en com usar el llenguatge scripting per a la implementació d'un contracte.

Un cop elaborat l'anàlisi, i fins la data d'entrega prevista de la tercera PAC es dissenyarà i implementarà un smart contract. Aquesta primera versió del contracte possiblement sigui funcional però no estarà provada ni verificada. Aquestes tasques es realitzaran fins a la finalització del projecte.

Per últim, la redacció i elaboració de la memòria està prevista que es realitzi durant l'execució d'aquest treball mentre que els últims dies es reserven per a l'elaboració de la presentació.

A la Figura 1.1 podem veure el diagrama de Gantt que ens mostra un resum de tota la planificació.

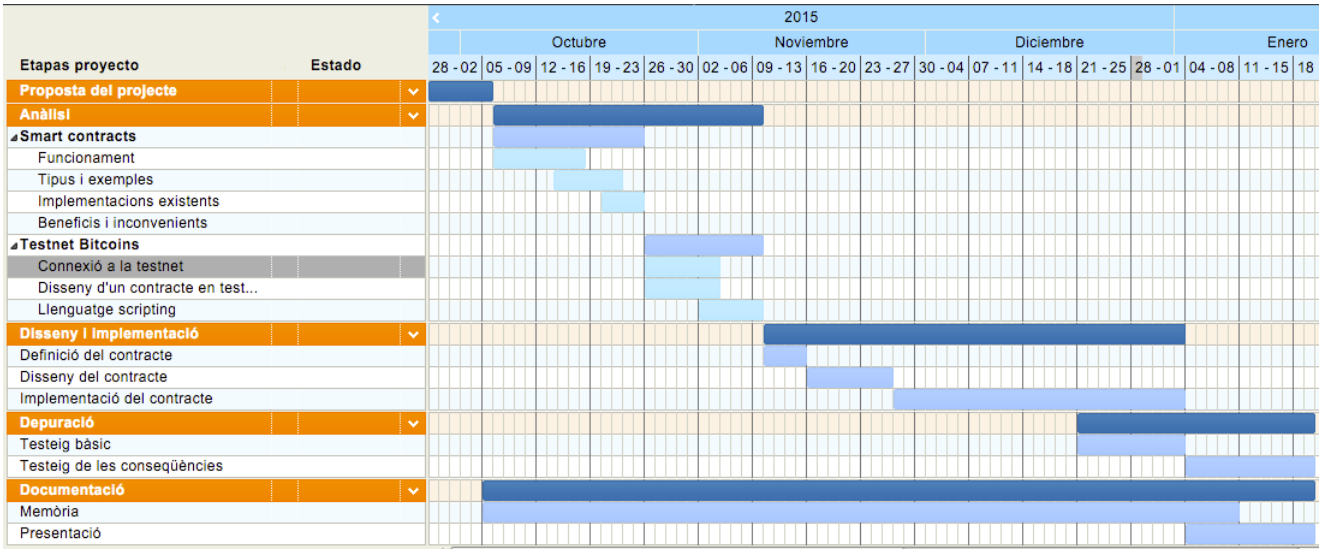

Figura 1.1: Diagrama de Gantt

CAPÍTOL 2

Smart Contracts

El concepte de *smart contract*[6][9] poc a poc va guanyant terreny dins el món tecnològic. Tot i així, no és un concepte senzill de definir. Peter Todd, un dels desenvolupadors del core de Bitcoin, el 5 de desembre de 2014 publicà el següent tuit:

> Conclusion of smart contract discussion: no-one has a clue what a smart contract actually is, and if we did it'd need oracles.

En aquest tuit, Peter opina que ningú té idea sobre què és realment un smart contract. Això ja denota una complexitat molt elevada per entendre que és un contracte intel·ligent, que pot abarcar o quina validesa pot tenir.

Per altra banda, Peter puntualitza que si es realitzés un contracte intel·ligent, necessitaríem oracles. En aquest cas, quan Peter parla d'oracles fa referència a un servidor que pugui signar transaccions d'una petició quan un usuari proveeix una expressió vertadera. Aleshores, el que ens vol dir en Peter és la necessitat de que el contracte pugui ser executat en el moment de disposar de la informació necessària.

En aquest capítol entrarem dins el món dels contractes intel·ligents, explicant els seus usos i donant alguns exemples.

2.1 El terme *smart contract*

Els contractes intel·ligents neixen de la idea de crear contractes amb la capacitat de fer-se complir a ells mateixos. Aquest concepte va ser proposat l'any 1994 per Nick Szabo. En general, per fer complir un contracte necessitem una o més persones o entitats externes imparcials que verifiquin

i/o executin el contracte. Només cal pensar en el cas d'una herència. És necessària l'actuació d'un notari per a validar el testament i executar-lo. Fins i tot necessitem altres entitats com poden ser els bancs segons el que contengui l'herència.

Aleshores es considera que el terme *smart contract* fa referència a qualsevol contracte que s'executa per sí mateix automàticament sense necessitat que tercers intervinguin entre els participants del contracte. És clar que aquest terme és molt obert i dóna lloc a moltes possibilitats sobre què considerar contracte i què no.

Per limitar l'abast d'aquest terme, considerem a més que aquests contractes són escrits com a un software, prescindint del llenguatge legal. Aquest software ha de ser capaç de definir regles i conseqüències estrictes, de la mateixa forma que quedaria especificat en un contracte tradicional. A més, pot prendre informació com a input per a processar-la segons les regles definides i actuar en conseqüència.

Després de les consideracions anteriors, hom pot preguntar-se com podem garantir que la informació que processa el contracte és fiable? Si tornem a l'exemple de l'herència, el notari rep documents certificats on garanteixen la defunció de la persona. En el cas del software, com es garanteix que l'autenticitat de la informació? Aquest és un gran repte que tenen els *smart contracts* i que han de resoldre per a realment ser fiables.

2.2 Usos i exemples

Els usos dels contractes intel·ligents no només es limiten a automatitzar alguns contractes tradicionals, sinó que obren portes a nous usos, redueixen despeses i intenen oferir garanties. Observem alguns casos d'ús de contractes intel·ligents:

- **Dipòsits de garantia**: un dipòsit de garantia és la recepció d'uns diners en concepte de garantia per si s'incompleix alguna condició d'algun acord. Un contracte intel·ligent perfectament es podria usar per a aquesta finalitat. Imaginem el cas d'una relació arrendador - arrendatari. Habitualment l'arrendatari ha de deixar en garantia una quantitat de diners. Un contracte intel·ligent podria ser capaç de detectar més a més que hi ha una transacció de l'arrendatari cap a l'arrendador amb la quantitat pactada. Si no és així, ell mateix transferiria el dipòsit de garantia a l'arrendador.

- **Herències**: imaginem que una persona té una certa quantitat de diners i que quan mori, vol que aquests automàticament els rebi el seu hereu. Informant de la mort de la persona en un lloc públic i creïble, el contracte intel·ligent podria realitzar la transacció sense necessitat que un notari i un banc intervinguin en la transacció.

- **Crowdfunding**: els últims anys ha sorgit una nova forma de recaptar fons cada cop més popular. El crowdfunding o micromecenatge consisteix en aconseguir donacions col·lectives per a un propòsit determinat. Habitualment s'especifica una quantitat mínima de diner a la qual arribar per portar a terme el projecte. Un cop arribat a realitzar el projecte, els mecenes poden rebre o no recompensa. En l'àmbit dels contractes intel·ligents, es podrien automatitzar molts processos fàcilment, com per exemple si no s'arriba a la quantitat de

diners desitjada, revertir les transferències o imposar mesures de seguretat perquè qui rebi els diners no els malversi.

- **Préstecs**: És molt habitual demanar préstecs per adquirir bens determinats com cotxes, ordinadors, etc. A part dels interessos elevats que es generen els préstecs, hi ha tot un procés molt llarg i problemàtic si el deute no es paga. Els contractes intel·ligents es podrien usar per bloquejar dispositius en cas d'impagament, impedir accessos etc. Fins que no s'aboni la quantitat de diners que es deu.
- **Loteries i apostes**: amb els contractes intel·ligents podrien néixer nous sistemes de loteries i apostes descentralitzats (avui en dia sempre hi ha d'haver-hi una entitat central per gestionar la recepció i el repartiment de les apostes).

2.3 Avantatges i inconvenients

Amb els exemples que hem vist abans, podem enumerar molts avantatges de la implementació de contractes intel·ligents: Els usos dels contractes intel·ligents no només es limiten a automatitzar alguns contractes tradicionals, sinó que obren portes a nous usos, redueixen despeses i intenten oferir garanties. Observem alguns casos d'ús de contractes intel·ligents:

- **Automatització**: moltes tasques manuals que requereixen que persones o tasques programades executin el contracte no són necessàries en un contracte intel·ligent ja que ell mateix s'encarrega de l'execució.
- **Rapidesa**: L'automatització implica a més una major eficiència i més rapidesa en l'execució dels contractes.
- **Absència de terceres entitats**: Amb l'automatització de contractes i les garanties que ofereixen, terceres entitats que fins ara interferien en els contractes poden desaparèixer, i amb conseqüència tot el contracte d'inici a fi només queda entre els interessats.
- **Disminució de despeses**: L'absència de terceres entitats inclou una disminució de despeses considerables que beneficia a totes les parts dels contractes.

Si bé els avantatges dels contractes intel·ligents són molt interessants, hi ha una sèrie d'inconvenients i problemàtiques molt importants que s'han de tenir en compte a l'hora de realitzar un smart contract:

- **Accés i fiabilitat de les dades d'entrada**: un contracte intel·ligent a l'executar-se automàticament ha de ser capaç d'adquirir fonts fiables de dades per a no cometre errors ni fraus.
- **Llenguatge legal**: el llenguatge legal és molt ambigu i implica que un contracte legal pugui tenir més d'una interpretació. És clar que un software no es pot permetre aquestes interpretacions. Si bé això pot semblar un avantatge, en el món real és difícil poder especificar clarament sense ambigüitat totes les condicions d'un contracte.

- **Acceptació**: el món legal evoluciona lentament i és molt difícil aconseguir l'acceptació jurídica dels contractes intel·ligents per part d'aquest món.
- **Anul·lacions i modificacions**: és molt habitual en un contracte voler canviar les condicions actuals o fins i tot anul·lar-lo si totes les parts estan d'acord. Els contractes intel·ligents han de preveure totes aquestes casuístiques per a realment ser una alternativa real als contractes.

2.4 Implementacions existents

Actualment hi ha pocs exemples de contractes intel·ligents implementats. Seguidament es presenten tres exemples: intercanvi d'arxius en P2P, el DRM i contractes sobre blockchain. Si bé l'intercanvi d'arxius P2P en sí no ha estat concebut com a contracte intel·ligent, és interessant recordar-lo ja que marca un precedent.

La finalitat del treball no és arribar al detall d'implementació d'aquests contractes, per la qual cosa només s'expliquen les nocions bàsiques d'aquests. En el cas de contractes sobre blockchain, s'aprofundirà en el tema al següent capítol, acotant aquest tipus amb el Bitcoin[8].

2.4.1 Intercanvi d'arxius en P2P

Per a moltes persones és conegut l'intercanvi d'arxius per una xarxa P2P. Programaris com Kazaa, BitTorrent, eMule o Ares van contribuir en la seva popularització i van obrir el gran debat sobre la pirateria que encara està present avui en dia.

Recordem que una xarxa P2P es basa en la connexió entre nodes, sense necessitat que hi hagi l'existència d'un servidor central on resideixi la informació. El funcionament de la majoria de softwares esmentats anteriorment que un usuari connectat a la xarxa P2P busqués permetia contingut que podia residir en 1 o més nodes de la xarxa i transferir-lo al seu ordinador. A canvi, l'usuari passava a ser un node distribuïdor més d'aquest contingut. Alguns softwares permetien limitar aquest rol de distribuïdor, mentre que d'altres n'era obligatori.

Aquest sistema de funcionament es pot considerar un precedent dels contractes intel·ligents. Estem davant un software que sense necessitat de tercers, quan un usuari decideix obtenir un contingut aquest passa a actuar com a un node més compartint la informació. La finalitat és aconseguir una xarxa on no només hi hagi consumidors de recursos, sinó que com a consumidor, també mantinguis la xarxa i passis a ser un distribuïdor d'informació.

Avui en dia les xarxes P2P han evolucionat molt i són més que una simple forma de compartir arxius. Per exemple, les criptomonedes descentralitzades s'han nodrit d'aquest sistema per a aconseguir ser realment distribuïdes, cosa que algunes d'elles han acabat oferint sistemes per a la creació de contractes intel·ligents.

2.4.2 El DRM

DRM són les sigles de "*Digital rights management*", és a dir, gestió de drets digitals. És un sistema molt conegut pels lectors de llibres digitals o els consumidors de música digital. La

finalitat d'aquest sistema és limitar l'accés al material comprat. D'aquesta forma, el venedor pot controlar qui i com accedeix al material.

Aquest sistema de limitació d'accés al contingut multimèdia es pot considerar com a un contracte intel·ligent. El contracte és entre el venedor i el comprador, on el comprador accepta que només podrà accedir a l'arxiu des d'un o més dispositius acordats.

Habitualment el software DRM vincula l'arxiu obtingut amb una combinació de hardware, software i usuari identificat. D'aquesta forma, quan s'intenta accedir a l'arxiu des del hardware vinculat, el DRM permet l'accés. De la mateixa forma, habitualment es permet l'accés des de software on prèviament l'usuari s'hagi hagut d'identificar. En canvi, el software del DRM si detecta que l'accés es produeix des d'un hardware diferent, bloqueja l'accés. Seria un incompliment del contracte entre comprador i venedor.

Com es pot veure, el DRM no necessita una intervenció de tercers per a identificar si es compleix o no la condició contractual. A més, és veu clarament que la identificació del hardware / software d'accés és un input al DRM i aquest actua en conseqüència: permetre l'accés o no a l'arxiu.

Cal remarcar que avui en dia hi ha molts softwares que han aconseguit trencar els sistemes DRM per donar lliure accés als arxius dels quals el venedor en vol limitar l'accés. Això no implica que el DRM no pugui ser considerat un contracte intel·ligent, sinó que tercers han estat capaços de trencar la seguretat d'aquest sistema convertint-lo en poc fiable per a ser usat.

Hi ha moltes empreses que proveeixen continguts amb aquest sistema: Apple, Sony, Amazon, Adobe o Microsoft entre d'altres. Aquestes usen el DRM en continguts propis com pot ser Apple amb la música de iTunes o Amazon amb els seus ebooks Kindle.

2.4.3 Contractes sobre blockchains

Aquests tipus de contractes intel·ligents segurament són els més coneguts avui en dia i és on la majoria d'experts hi dediquen els seus esforços. Gràcies al naixement del Bitcoin com a moneda descentralitzada, es van introduir nous elements i eines com les blockchains[7] (cadenes de blocs) que ara són usades per a la creació de contractes intel·ligents.

En totes les monedes basades en blockchain, tots els nodes que formen part de la xarxa corresponent mantenen una llista comuna de totes les transaccions conegudes. Això és el que s'anomena cadena de blocs. Els nodes generadors de la moneda (anomenats també miners) són qui creen els blocs afegint un hash de l'últim bloc creat de la cadena més llarga i de les noves transaccions acumulades des de la creació de l'últim bloc. Quan un miner troba un bloc, el comparteix amb tota la resta de nodes als que està connectat.

Aquesta forma de comunicar transaccions sense necessitat d'una entitat central, permet la realització de transaccions distribuïdes entre usuaris. A més es garanteix que no hi hagi doble despesa de la moneda per part dels usuaris. Gràcies a aquest funcionament, i a la dotació d'opcions per a la generació de scripts (més o menys potents segons la moneda) neixen els contractes intel·ligents sobre blockchains.

El Bitcoin incorpora un llenguatge scripting que permet realitzar algunes accions, permetent així implementar contractes intel·ligents. Altres monedes aprofiten la infraestructura de Bitcoin

directament afegint capes per sobre i d'altres han generat una nova cadena de blocs modificant el protocol original de Bitcoin. Dins aquestes últimes monedes cal destacar Ethereum, que començà a engendrar-se al desembre de 2013 i publicada finalment a l'hivern del 2014-15.

CAPÍTOL 3

Transaccions en Bitcoin

El sistema de bitcoins és una moneda electrònica que fou concebuda per una persona, o grup de persones, amb el pseudònim de Satoshi Nakamoto. La xarxa P2P sobre la qual es sostenen entrà en funcionament al gener del 2009. El seu anonimat i la mancança d'una entitat central l'han convertit en una moneda molt interessant. De fet, ha marcat precedents en la concepció de noves monedes digitals. La majoria de les noves monedes que s'han creat aquests últims anys estan basades en el protocol que utilitza Bitcoin.

L'ús més conegut del sistema de bitcoins és el funcionament com a moneda digital. Tot i així, el sistema va més enllà. El seu sistema blockchain ha estat usat per a concebre contractes intel·ligents descentralitzats. Actualment, el protocol Bitcoin ofereix un llenguatge scripting que permet la creació d'aquests contractes. L'objectiu d'aquest capítol és entendre el funcionament de les transaccions i el llenguatge scripting, per a posteriorment veure les possibilitats que ens ofereix Bitcoin per a la generació de contractes intel·ligents. Es pressuposa que el lector té els coneixements bàsics del funcionament de Bitcoin.

3.1 Les transaccions

Per entendre les transaccions del sistema Bitcoin[1][4], ajuda pensar-les de la mateixa forma que un xec de paper. Una transacció és la forma en que s'expressa la intenció de transferir diners i no és visible per tot el sistema fins que es presenti a un node per a la seva execució.

El cicle de vida de la transacció comença amb la creació d'aquesta. Posteriorment és firmada amb una o més signatures que són les que l'autoritzen. Seguidament, la transacció es transmet per tota la xarxa Bitcoin on cada node la valida i la propaga als seus nodes coneguts.

Quan aquesta arriba a un node miner, aquest la verifica i la inclou en un bloc de transaccions, que posteriorment queda inclòs a la blockchain. Si la transacció no és verificada, el node miner la descarta.

D'aquesta forma acaba el cicle de vida d'una transacció. En aquest punt tots els nodes participants de la xarxa accepten la transferència i els bitcoins poden ser gastats de nou. Tot i així, per seguretat en cas de que sorgeixin branques a la blockchain, es demana que s'hagin generat alguns blocs més posteriors al que inclou la transacció.

Com hem comentat abans, una transacció pot ser creada en línia o fora de línia. Això permet que, a l'igual que els xecs, es generin vàries transaccions i es mantinguin a l'espera d'executar-se. A més dona peu a separar qui genera la transacció del qui la signa. Per exemple, habitualment si una empresa paga les nòmines amb xecs, els qui generen aquests xecs no són qui els signen. Amb les transaccions es pot fer exactament el mateix.

3.1.1 Estructura de les transaccions

A nivell d'implementació, una transacció és una estructura de dades que codifica una transferència de bitcoins a partir dels fons d'un origen a una destinació. Aquests orígens i destinacions no estan relacionats amb comptes o identitats, de manera que la moneda no està físicament guardada en una adreça, sinó que el valor de bitcoins continguts en una adreça es calcula en base a les transaccions que ha rebut. Notem que les transaccions no estan xifrades, de forma que qualsevol persona pot consultar-ne el contingut.

Observem l'estructura essencial d'una transacció a la Taula 3.1.

Mida	**Camp**	**Descripció**
4 bytes	Version	versió actual de la transacció
1-9 bytes (VarInt)	Input Counter	Enter que ens indica el nombre d'entrades
Variable	Inputs	Una o més transaccions d'entrada
1-9 bytes (OutInt)	Output Counter	Enter que ens indica el nombre de sortides
Variable	Outputs	Una o més transaccions de sortides
4 bytes	Locktime	Marca de temps de UNIX o nombre de bloc

Taula 3.1: Estructura bàsica d'una transacció

Les dues parts bàsiques de la transacció són les entrades i les sortides. Com el seu nom indica, les entrades contenen la informació necessària de les adreces des d'on es volen moure els diners mentre que la sortida ens marca la destinació d'aquests. Com es pot veure en l'estructura, una sola transacció admet més d'una entrada i més d'una sortida. Això ens permet realitzar moviments entre vàries adreces en una sola transacció. En els següents apartats s'explica detalladament l'estructura d'aquestes transaccions.

El paràmetre locktime, també anomenat nLockTime, defineix a partir de quan la transacció serà vàlida i es podrà transmetre per la xarxa. Habitualment s'ajusta a 0 en la majoria de transaccions per a propagar-la immediatament. En cas de no fer-ho, només es transmetran quan siguin vàlides.

Per a referenciar les transaccions, totes elles estan identificades amb una id. Aquesta id és un doble hash SHA256 de tots els paràmetres de la transacció. Gràcies a aquesta id, podem referenciar les transaccions i així indicar la transacció d'origen quan volem gastar bitcoins.

3.1.2 Entrades d'una transacció

Una entrada d'una transacció es pot entendre com una referència a la sortida d'una altra transacció existent que ens autoritza gastar aquests diners. Observem l'estructura d'una entrada d'una transacció entrada a la Taula 3.2

Mida	Camp	Descripció
32 bytes	Transaction Hash	Hash de la transacció prèvia
4 bytes	Previous Txout-index	Enter que ens indica l'índex de la sortida
1-9 bytes (VarInt)	inputScript Size	Mida de l'script
Variable	inputScript	Script
4 bytes	Sequence Number	Camp deshabilitat

Taula 3.2: Estructura bàsica d'una entrada d'una transacció

Com es pot veure, una entrada conté un hash de la transacció prèvia mentre que Previous Txout-index ens indica l'índex de la sortida concreta de la transacció d'origen (recordem que una transacció pot tenir més d'una sortida).

Els següents paràmetres són un script i la seva longitud. El contingut d'aquest script consta d'una signatura i una clau pública, que pertany al propietari de la transacció d'origen. És la prova que el creador de la transacció està autoritzat a gastar aquesta quantitat de diners. Pel que fa la signatura, és una signatura digital ECDSA aplicada a un hash d'una versió simplificada de la transacció d'origen. Aquesta signatura té la finalitat de verificar que la transacció ha estat generada per l'autèntic propietari de la direcció d'origen. Aquest script inclou diverses instruccions que permeten realitzar diferents tipus de pagament, que s'explicaran més endavant.

Observem que a les entrades no s'especifica cap import. L'import a gastar l'obtenim buscant-lo a les sortides de la transacció a la que es fa referència amb el hash.

3.1.3 Sortides d'una transacció

Una sortida conté les instruccions necessàries per a enviar la suma de bitcoins. Observem l'estructura de les sortides a la Taula 3.3.

Mida	Camp	Descripció
8 bytes	Amount	Quantitat de bitcoins a transferir
1-9 bytes (VarInt)	outputScript Size	Mida de l'script
Variable	outputScript	Script

Taula 3.3: Estructura bàsica d'una sortida d'una transacció

Observem que hi ha un camp valor on s'inclou la quantitat de moneda desitjada que es vol fer arribar a la destinació. El valor no està representat amb BTC sinó en satoshis (1 BTC =

100 000 000 satoshis). Els altres dos paràmetres són un script i la seva longitud. Aquest script és el que s'encarrega de verificar que les entrades tenen autorització per transferir els bitcoins a les sortides.

Com s'ha comentat abans, pot haver-hi més d'una sortida. Entre elles s'han de repartir la suma total de bitcoins de l'entrada. Això significa que, per exemple, si tenim una quantitat de 20 BTC en una adreça i volem transferir-ne 10 a un altre usuari, haurem de crear una sortida per a que l'usuari rebi els 10 BTC i una segona per quedar-nos amb el canvi. Si no ho fem, perdríem el canvi.

També cal notar que és obligatori gastar l'import de la suma de totes les entrades, pero que en general la suma de les sortides en general no serà exactament la suma de les entrades. El motiu és la comissió que s'emporten els miners per al manteniment de la xarxa Bitcoin. Aquesta comissió pot fer que la nostra transacció tingui més o menys prioritat per a confirmar-la.

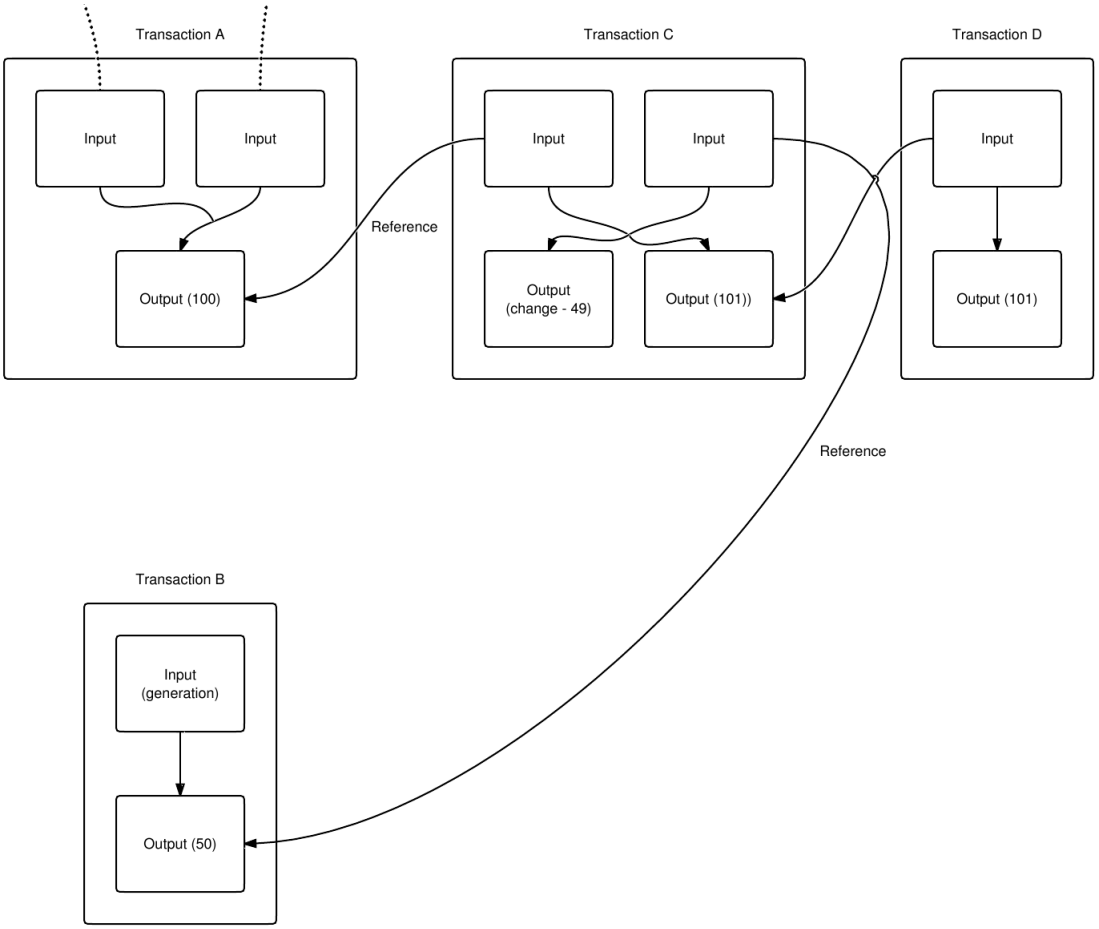

Figura 3.1: Referències entre entrades i sortides

3.1.4 Verificació de transaccions

Ja hem vist l'estructura de les transaccions i totes les dades que contenen. Com hem comentat, cada transacció inclou una sèrie d'sripts per a verificar les transaccions. Per fer-ho, l'script de

les transaccions de sortida outputScript avalua els valors retornats de l'script de les transaccions d'entrada inputScript. Si la verificació és correcta, es considera que els diners poden ser transferits.

Aquesta metodologia de verificació és la que obre les portes als contractes intel·ligents als bitcoins. El llenguatge script permet crear condicions complexes que al ser executades poden ser usades per a la generació de contractes intel·ligents.

En el següent apartat s'explicarà amb detall les bases i el funcionament d'aquest llenguatge scripting, posant d'exemple la verificació d'una transacció estàndard.

3.2 Llenguatge scripting

El llenguatge scripting dels bitcoins[2][4] està basat en un llenguatge anomenat Forth, sorgit als anys 60. És un llenguatge d'instruccions simples, basat en pila i processat d'esquerra a dreta. Es considera un llenguatge Turing no complet, sense l'opció d'executar bucles.

Com a tota pila, l'estructura de dades del llenguatge permet dues operacions: push i pop. Recordem que el push afegeix un element a la fila i el pop l'elimina de la part superior, és a dir, l'últim element apilat és el primer que eliminem.

L'intèrpret del llenguatge script processa les instruccions. Quan l'intèrpret troba un nombre, l'insereix a la pila amb un push. En canvi, quan es troba un operador / instrucció realitza un pop de tants elements com siguin necessaris. També és possible que un cop es realitza l'operació, es faci un push a la pila amb el resultat calculat. Aquest és el cas de l'operació OP_ADD, que obté dos elements de la pila, els suma i incorpora de nou el resultat a la pila.

En la majoria de transaccions que contenen scripts, es necessita obtenir un resultat TRUE per a que la transacció sigui vàlida. El llenguatge proveeix d'una sèrie d'instruccions condicionals per a poder obtenir un resultat TRUE o FALSE que s'acabi carregant a la pila.

Per entendre el funcionament bàsic del llenguatge, considerem la següent operació: 2 + 7 - 3 + 1. Utilitzant les instruccions OP_ADD per a la suma, OP_SUB per a la resta i OP_EQUAL per a comparar l'operació amb el número 7 (el resultat de l'operació), l'script seria els següent:

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

Observem com varia la pila en les diferents iteracions a la Taula 3.4. Com es pot veure, el funcionament essencial del llenguatge és molt senzill i es basa totalment amb les funcionalitats push i pop de la pila.

3.2.1 Scripts de transaccions estàndards

Segons el protocol, qualsevol altre tipus d'scripts diferent a aquests haurien de ser rebutjats pels nodes i per tant, la transacció no es podria confirmar.

Iter.	Pila	Script	Acció
1	-	`2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL`	Iniciem l'execució
2	2	`7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL`	Push 2
3	2, 7	`OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL`	Push 7
4	9	`3 OP_SUB 1 OP_ADD 7 OP_EQUAL`	Sumem els elements
5	9, 3	`OP_SUB 1 OP_ADD 7 OP_EQUAL`	Push 3
6	6	`1 OP_ADD 7 OP_EQUAL`	Restem els elements
7	6,1	`OP_ADD 7 OP_EQUAL`	Push 1
8	7	`7 OP_EQUAL`	Sumem els elements
9	7, 7	`OP_EQUAL`	Push 7
10	TRUE	-	Comparem els elements

Taula 3.4: Exemple de la pila d'un script

Anem a veure com es realitza l'execució d'un script d'una transacció estàndard. Recordem que hi ha un script a les transaccions d'entrada (inputScript) i un script a les transaccions de sortida (outputScript). Per a aquestes transaccions estàndard l'inputScript s'acostuma a referenciar com a scriptSig i l'outputScript com a scriptPubKey per les raons que quedaran clares en l'explicació. Un possible script seria el següent:

`scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG`

`scriptSig: <sig> <pubKey>`

On els elements sig, pubKey i pubKeyHash són valors numèrics.

En primer lloc, s'uneixen els scripts scriptSig i scriptPubKey, de manera que el primer en executar-se és scriptSig. Aleshores, les iteracions de l'execució serien els que veiem a la Taula 3.5.

Com hem vist, el llenguatge scripting ofereix totes les eines necessàries per a la verificació de transaccions: càlculs de hash, comparadors de signatures, etc. És més, el llenguatge ofereix un conjunt d'instruccions molt ampli que ens permeten crear scripts molt més complicats que el que acabem de veure.

Tot i la possibilitat de crear scipts més complicats, actualment la xarxa bitcoin només accepta transaccions amb tres tipus d'scripts diferents (deixant de banda les transaccions de generació):

- El tipus P2PKH (Pay-to-PubkeyHash és l'usat per a les transaccions estàndard. És el tipus d'script més habitual i és el que acabem de veure.
- El tipus P2PK és molt semblant a l'anterior però actualment està en desús, ja que ocupen més espai que els P2PKH.
- P2SH (Pay-to-Script-Hash) són uns tipus d'scripts que permeten realitzar transaccions on siguin necessari que un o més usuaris signin la transacció per a poder incloure-la dins d'un bloc.

Iter.	Pila	Script	Acció
1	-	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Iniciem l'execució
2	<sig>	<pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Iniciem l'execució
3	<sig>, <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Afegim les constants a la pila
4	<sig>, <pubKey>, <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Dupliquem l'últim element
5	<sig>, <pubKey>, <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Obtenim el hash de l'últim element
6	<sig>, <pubKey>, <pubHashA>, <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Afegim una constant
7	<sig>, <pubKey>	OP_CHECKSIG	Comparem els dos últims elements
8	TRUE	-	Es comprova la signatura dels dos elements superiors de la pila

Taula 3.5: Exemple de la pila d'una transacció estàndard

3.2.2 Repertori d'instruccions

Per a poder visualitzar el potencial real d'aquest llenguatge, presentem part del repertori d'instruccions. Com es veurà, el llenguatge ofereix vàries instruccions pensades totalment per a la manipulació de dades criptogràfiques i validació de transaccions.

En primer lloc, observem les instruccions de control de flux. Aquestes engloben condicionals i validació de transaccions a la Taula 3.6. L'estructura dels blocs condicionals és:

```
<expression> if [statements] [else [statements]]* endif
```

Seguim amb les instruccions aritmètiques a la Taula 3.7. Aquestes estan limitades a enters amb signe de 32 bits, però es permet overflow en la sortida. Si s'utilitzen valors de mida superior per a realitzar operacions, l'script queda avortat i fallarà.

Observem que no hem inclòs instruccions producte i divisió. Tot i que existeixen, es mantenen deshabilitades i per tant no poden ser usades en les transaccions bitcoin.

Paraula	**Entrada**	**Sortida**	**Acció**
OP_NOP	-	-	no realitza cap acció
OP_IF	**	**	condicional if, que s'executa si l'últim element de la pila és diferent de 0
OP_NOTIF	**	**	igual que l'if, però s'executa si el valor és 0
OP_ELSE	**	**	equivalent al "sinó"
OP_ENDIF	**	**	indica la finalització d'un bloc condicional
OP_VERIFY	boolean	-	si al top de la pila és true, marca la transacció com a vàlida, en altre cas l'invalida
OP_RETURN	-	-	marca la transacció com a no vàlida

Taula 3.6: Scripting. Instruccions condicionals

Un altre tipus d'instruccions molt interessants són les criptogràfiques. Aquestes ens permeten treballar amb les claus i els hash de les adreces bitcoin de forma senzilla.

Hi ha moltes més instruccions que les llistades en aquest apartat. Les funcionalitats inclouen manipulacions de la pila (com duplicar elements, eliminar-ne, etc.) o operadors lògics.

Paraula	Entrada	Sortida	Acció
OP_1ADD	in	out	sumem 1 a l'entrada
OP_1SUB	in	out	restem 1 a l'entrada
OP_NEGATE	in	out	canviem el signe de l'entrada
OP_ABS	in	out	calculem el valor absolut de l'entrada
OP_NOT	in	out	si l'entrada és 1 o 0, els intercanvia, en altre cas retorna 0
OP_0NOTEQUAL	in	out	retorna 0 si l'entrada és un 0, en altre cas retorna 1
OP_ADD	a b	out	operació suma a+b
OP_SUB	a b	out	operació resta a-b
OP_NUMEQUAL	a b	out	retorna 1 si a=b, en altre cas retorna 0
OP_NUMEQUALVERIFY	a b	out	verifica la transacció si a=b, en altre cas la invalida

Taula 3.7: Scripting. Instruccions aritmètiques

Paraula	Entrada	Sortida	Acció
OP_SHA256	in	hash	retorna el hash SHA256 de l'entrada
OP_HASH160	in	hash	calcula el hash SHA256 de l'entrada i després el RIPEMD-160
OP_CODESEPARATOR	-	-	separador de paraules entre transaccions
OP_CHECKSIG	sig pubkey	boolean	calcula el hash amb les dades de les transaccions fins a trobar un separador. Si la signatura és vàlida retorna 1 sinó 0
OP_CHECKSIGVERIFY	sig pubkey	boolean	el mateix que OP_CHECKSIG per però a més executa OP_VERIFY

Taula 3.8: Scripting. Instruccions criptogràfiques

CAPÍTOL 4

Implementació d'un canal de micropagaments

Els canals de micropagament estan pensats per a realitzar diversos pagaments d'imports baixos. El principal obstacle d'aquests pagaments són les comissions, ja que els converteixen en poc rentables. D'aquí sorgeix la necessitat d'implementar noves formes de gestionar aquests pagaments.

En els capítols anteriors s'han mostrat les eines necessàries per a la creació d'un contracte intel·ligent sobre bitcoin[5]. En aquest capítol s'explicarà la implementació d'un contracte intel·ligent sobre bitcoin usant aquestes eines per a la creació d'un canal de micropagaments.

En primer lloc es detallarà el funcionament de les transaccions P2SH i seguidament el disseny del protocol de micropagaments. Per últim, es veurà com s'ha implementat el canal i la forma d'utilitzar-lo.

4.1 Les transaccions P2SH

En l'Apartat 3.2.1 hem vist el funcionament de les transaccions estàndard. Recordem que aquestes transaccions només requereixen d'una signatura per a ser confirmades. Però que succeeix si desitgem que hi siguin necessàries dues o més signatures per a que la transacció sigui vàlida?

Les transaccions i adreces P2SH ens proporcionen una forma de solucionar aquesta mancança. Introduïdes a l'hivern del 2012 al BIP0016, les adreces P2SH impliquen l'ús de diferents adreces estàndard amb les seves respectives claus. Aleshores, per a realitzar transaccions des d'aquestes adreces són necessàries una o vàries signatures usant les respectives claus privades.

Per a crear una transacció per a gastar els bitcoins d'una adreça P2SH, s'utilitza el següent script:

```
0 <sig1> <sigM> OP_M <pubKey1> <pubKeyN> OP_N OP_CHECKMULTISIG
```

On M és el nombre de signatures necessàries per a signar la transacció i N el nombre total de clau públiques utilitzades per a generar l'adreça. La resta de paràmetres de l'adreça s'utilitzen igual que les transaccions estàndard.

Aquest script comentat ens permetrà crear el nostre contracte intel·ligent per a la implementació del canal de micropagaments.

4.2 El protocol

El protocol de del canal de micropagaments[3] es basa en tres punts clau:

- Transaccions P2SH que consten de dues claus públiques i es requereix que signin amb les dues claus privades respectives.
- Bloqueig de transaccions amb el paràmetre nLockTime
- Transaccions fora de línia (és a dir, transaccions de les quals no se'n fa broadcast)

El protocol aplicat consta d'un de tres passos d'inicialització i d'un seguit de passos iteratius fins a finalitzar el canal. La Taula 4.1 ens mostra l'esquema del protocol, on Alice és qui a paga 100 BTC a Bob mitjançant micropagaments amb data de finalització del canal del 15/03/2016.

Podem suposar que prèviament al protocol, Alice i Bob s'han intercanviat les claus públiques de les adreces A i B. A més, per simplificar-ho s'han obviat les comissions als miners de les transaccions.

	Alice		Bob
-2.	Create a transaction: $\text{Unsig}_A(\text{Tx}_0) = \{\text{in:}Adr_{ini}; \text{out:}Adr_{AB}; 100\ \text{BTC}\}$ Sign $\text{Unsig}_A(\text{Tx}_0)$: $\text{Tx}_0 = \{\text{in:}Adr_{ini}; \text{out:}Adr_{AB}; 100\text{BTC}; sig_A(\text{Tx}_0)\}$ Create a transaction: $\text{Unsig}_{AB}(\text{Tx}_{seg}) = \{\text{in:}Adr_{AB;}\text{out:}Adr_A; 100\ \text{BTC}; \text{Exp: } 2016/03/15\}$	$\xrightarrow{\text{Unsig}_{AB}(\text{Tx}_{seg})}$	
-1.		$\xleftarrow{\text{Unsig}_A(\text{Tx}_{seg})}$	Sign $\text{Unsig}_{AB}(\text{Tx}_{seg})$: $\text{Unsig}_A(\text{Tx}_{seg}) = \{\text{in:}Adr_{AB;}\text{out:}Adr_A; 100\text{BTC}; \text{Exp: } 2016/03/15; sig_B(\text{Tx}_{seg})\}$
0.	Sign $\text{Unsig}_A(\text{Tx}_{seg})$: $\text{Tx}_{seg} = \{\text{in:}Adr_{AB;}\text{out:}Adr_A; 100\text{BTC}; \text{Exp: } 2016/03/15; sig_B(\text{Tx}_{seg}); sig_A(\text{Tx}_{seg})\}$ Store Tx_{seg} Broadcast Tx_0 (from step -2) to be included in the blockchain		
i.	Create the transaction: $\text{Unsig}_{AB}(\text{Tx}_i) = \{\text{in:}Adr_{AB}; (\text{out}_1 : Adr_A; (100 - (i \cdot k))\text{BTC}), (\text{out}_2 : Adr_B, (i \cdot k)\text{BTC})\}$ Sign the transaction: $\text{Unsig}_B(\text{Tx}_i) = \{\text{in:}Adr_{AB}; (\text{out}_1 : Adr_A; (100 - (i \cdot k))\text{BTC}), (\text{out}_2 : Adr_B, (i \cdot k)\text{BTC}); sig_A(\text{Tx}_i)\}$	$\xrightarrow{\text{Unsig}_B(\text{Tx}_i)}$	
i^*.			Sign $\text{Unsig}_B(\text{Tx}_i)$ Store Tx_i
n.			Broadcast Tx_n to be included in the blockchain

Taula 4.1: Protocol del canal de micropagaments

Veiem amb detall tots els passos del protocol:

- Pas -2: Alice crea una transacció T_{x_0} on transfereix 100 BTC a l'adreça AB i la signa però no en realitza un broadcast. Aquesta adreça és del tipus P2SH i requerirà les signatures tant d'Alice com Bob per extreure'n els diners.

  Seguidament Alice crea una segona transacció anomenada $T_{x_{seg}}$. Aquesta transacció gasta T_{x_0} i retorna tots els bitcoins a l'adreça A. És una transacció de seguretat que Alice pot usar en cas que hi hagi problemes amb el Bob i pugui recuperar-ne els bitcoins. $T_{x_{seg}}$ inclou a més una data de bloqueig (usant el paràmetre nLockTime), de forma que Alice no podrà realitzar un broadcast de la transacció fins que s'arribi a aquesta data.

  Per últim, Alice signa $T_{x_{seg}}$ i l'envia al Bob.

- **Pas -1**: Bob rep $T_{x_{seg}}$ i en verifica el seu contingut. Si tot és correcte, signa la transacció i la retorna a l'Alice. D'aquesta forma, Alice disposarà de la transacció signada que podrà usar-la si hi ha algun problema.

- **Pas 0**: Alice signa $T_{x_{seg}}$, la guarda i realitza un broadcast de T_{x_0}. Arribats a aquest pas, els bitcoins estan en una adreça compartida que requereix tant la signatura de l'Alice com del Bob per a ser gastada.

- **Passos iteratius** i: Un cop inicialitzat el protocol, Alice està en disposició de realitzar micropagaments al Bob. El procés consistira que per cada micropagament, Alice generi una transacció T_{x_i}, on la segona sortida consta del micropagament a Bob i la primera del canvi d'Alice. Recordem que aquesta transacció requereix que sigui signada per tots dos.

  Alice un cop crea la transacció, la signa i l'envia a Bob. Bob en revisa el contingut, la signa i la guarda però no en realitza Broadcast. En aquest pas, Bob ja disposa d'una transacció que li permetrà obtenir el pagament en qualsevol moment.

  Si suposem que l'Alice vol realitzar micropagaments d'una quantitat k, en cada iteració Bob rebrà transaccions de valor $i \cdot k$ mentre que l'Alice tindrà un canvi de $100 - i \cdot k$.

- **Pas final** n: Per a finalitzar el canal, el Bob realitza un broadcast de l'última transacció T_{x_i} rebuda (anomenem-la T_{x_n}) per a rebre a la seva adreça tots els micropagaments.

  Observem que cada transacció T_{x_i} acumula l'anterior. Gràcies a aquesta implementació, Bob pot anar acumulant els bitcoins però només ha de realitzar broadcast de la transacció T_{x_n}, de forma que només es pagarà un cop la comissió.

Notem que és molt important definir una data d'expiració del canal de micropagaments i que la transacció $T_{x_{seg}}$ tingui una data de bloqueig igual o superior. Si no fos així, l'Alice podria realitzar micropagaments al Bob i quan expirés la data de bloqueig de T_{x_i}, recuperar els bitcoins privant a Bob d'obtenir els micropagaments. A més, fixem-nos que en tot el protocol només és realitzen dos broadcast. Per tant, només es pagaran dues comissions tot i que es realitzin n micropagaments.

4.3 Implementació en Python

Per a la implementació del contracte intel·ligent explicat, una forma senzilla i fàcil d'entendre a nivell de codi és implementar un model client - servidor amb Python.

Els requisits i llibreries externes necessaris per a la implementació són:

- Python 3.x
- Llibreria Python bitcoinlib[1]
- Llibreries de desenvolupador OpenSSL

La implementació pot funcionar tant a la xarxa bitcoin com a la xarxa testnet, indicant per paràmetre la xarxa utilitzada. És important que tant client com servidor s'utilitzin amb la mateixa xarxa. D'aquesta forma podem realitzar transaccions a la testnet sense que suposi un cost.

Anem ara explicar genèricament el funcionament de la implementació. El detall del codi s'adjunta a l'Apèndix A.

4.3.1 Client

El client permet realitzar bàsicament tres funcionalitats:

1. Inicialitzar el protocol: intercanvia les claus públiques amb el servidor, crea la transacció de pagament inicial a l'adreça compartida, crea la transacció de seguretat i espera la signatura del servidor.
2. Realitzar micropagament: genera una transacció de micropagament
3. Realitzar boradcast de la transacció de seguretat: permet realitzar broadcast de la transacció de seguretat

Per a la transacció inicial és necessari disposar de la clau privada d'una adreça que contingui bitcoins. Per passar-li la clau privada es pot fer tant per paràmetre en la crida del programa com per un fitxer encriptat usant l'algorisme AES. La primera opció és adequada per a realitzar proves però és totalment desaconsellable en un entorn real. Pel que fa el broadcast, el mateix programa el realitza, sense necessitat d'usar eines externes.

Per a evitar l'execució permanent del client, en el moment de la inicialització del protocol es genera un fitxer JSON on es guarda tota la informació necessària per a realitzar els posteriors passos. Per seguretat, el fitxer generat s'encripta per a evitar ser llegit per tercers.

La comunicació amb el servidor es fa amb sockets TCP/IP de la llibreria socket de Python 3. Pel canal només es transmeten claus públiques i transaccions sense signar. Per tant no és problemàtic que les dades siguin transparents.

[1]Disponible al repositori petertodd / python-bitcoinlib de GitHub

4.3.2 Servidor

El servidor sempre està en execució i no requereix de cap acció externa. Aquest sempre està escoltant esperant rebre la inicialització del protcol per part d'un client i emmagatzemar transaccions rebudes. En cas de rebre altres instruccions, el servidor les ignorarà per complet.

L'execució del servidor finalitzarà quan s'arribi al temps de duració establert al canal de micropagaments o quan s'hagi arribat a un objectiu de bitcoins fixat. En aquest moment el servidor realitzara broadcast de les últimes transaccions dels clients.

Igual que el client, el servidor necessita generar fitxers JSON per guardar les dades dels clients que han interactuat amb ell. Aquests fitxers poden estar encriptats amb l'algorisme AES per evitar lectures de tercers. Per a identificar els clients, s'utilitza la clau pública, ja que aquesta és única i identifica fàcilment el client amb el que s'està interactuant.

4.4 Proves

Per a verificar el funcionament de la implementació del canal s'han realitzat diverses proves sobre la testnet de bitcoin. En aquesta memòria en documentarem dues: el cas en el qual el servidor arriba a l'objectiu definit i un cas en el qual el servidor deixa d'estar disponible i el client recupera el seu pagament.

4.4.1 Cas 1: objectiu complert

En primer lloc, s'ha inicialitzat el servidor amb els paràmetres de la Figura 4.1 i el client amb els paràmetres que es visualitzen a la Figura 4.2. Per inicialitzar el client, prèviament hem creat un arxiu encriptat que conté la clau privada de l'adreça que conté els bitcoins.

```
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$ ./server.py -port 9999 -obje
ctiu 0.06 -duracio 4 -password misticuoc -broadcast 1
Nou donatiu. BTC actual: 0.01
Nou donatiu. BTC actual: 0.02
Nou donatiu. BTC actual: 0.03
Nou donatiu. BTC actual: 0.04
Nou donatiu. BTC actual: 0.05
Nou donatiu. BTC actual: 0.06
S ha arribat a l objectiu. Broadcast de totes les transaccions.
01000000016a33e447e878430f31a089af42ab5569744ad0485d7410c7f807d2c6716b7762000000
00db00483045022100f47002ecfda88f4d40e795950900f692a4cc272ee6402fe1744916b05bfd52
c502203605e5eac1be40a3fbd98e4ce004d074a1b1b8f1ae591f461cebbf37430853f60148304502
2100bdcc8acae698f1a718bce00df68bfd9e6d3e8e01ceff72f071ad0128943f8b38022023a105c1
3c514c7ae665d460986b6a4c7f00d63f37ba6a640772545c1b87d5a80147522103ab40bb1b712c9f
75e41fc9e86f9f266eef9a152c54a7e8b2082969494aa1aabe21035ca5cbb49d689f40b7b7c51896
4fcec2f14e7c1e695db985e7923613520ba4f752aeffffffff0220f40e00000000001976a91468e0
709a2294d9975849a2035be95ba0392da0d088ac808d5b00000000001976a914537e59f94f857afc
e30dcf257a9983694d62ca7b88ac00000000
Resposta broadcast: {"txid":"fd45b7183d8b973a0264b83c9881f9215ffbc57b2c9cf23018f
3344f26559c19"}
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$
```

Figura 4.1: Inicialització i recepció de pagaments del servidor

Observem que la inicialització és correcte i el client realitza un broadcast de la transacció inicial, de manera que en aquest punt els bitcoins estan en una adreça que requereix que el client i el servidor es posin d'acord per extreure'ls. Aquesta transferència es pot veure a la Figura 4.3.

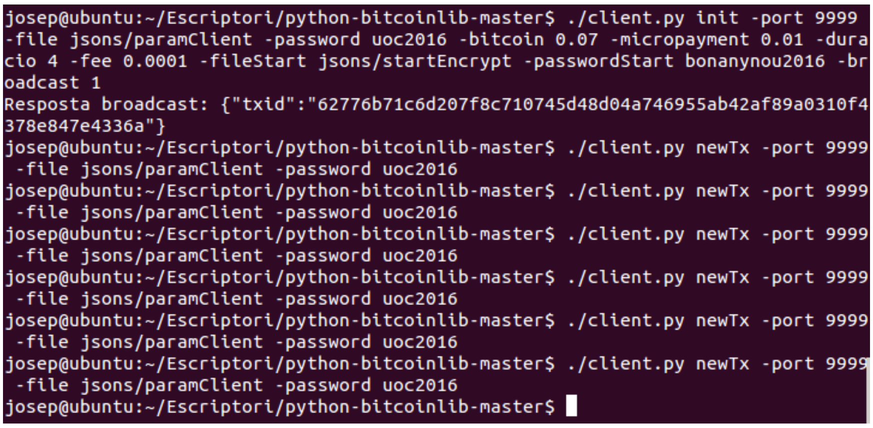

Figura 4.2: Inicialització i enviament de pagaments del client

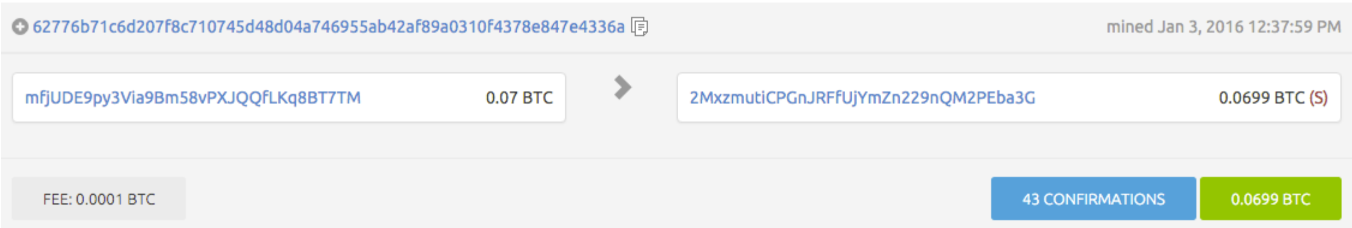

Figura 4.3: Transferència inicial

Un cop feta la transferència inicial, a la Figura 4.2 observem que el client va realitzant diferents micropagaments i aquests són rebuts pel servidor. Observem a la Figura 4.1 que el servidor quan arriba a l'objectiu, realitza un broadcast de l'última transacció rebuda. Per tant, aconseguim el desitjat: el client va realitzant micropagaments i el servidor quan ho decideix pot transferir-se els bitcoins, tornant el canvi al client. Aquesta última acció queda reflectida a la blockchain, com podem veure a la Figura 4.4. Notem a més que és l'única transacció de les que té guardades el servidor que s'ha afegit a la blockchain.

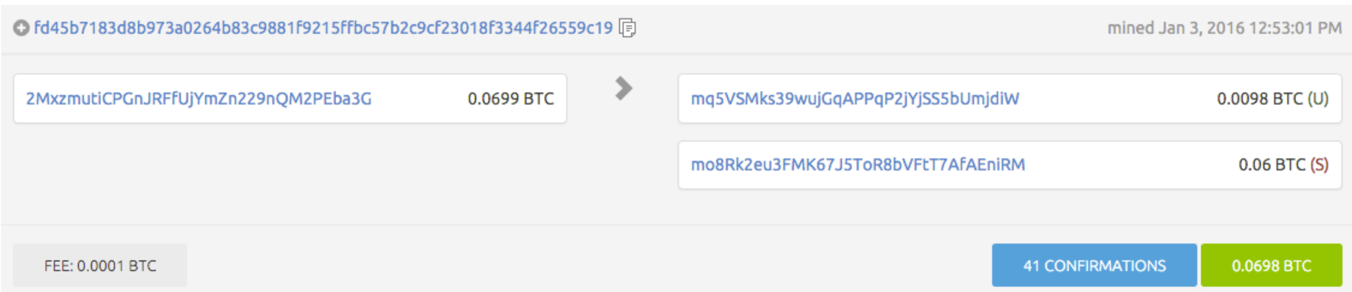

Figura 4.4: Transferència final

4.4.2 Cas 2: caiguda del servidor

Suposem ara que inicialitzem client i servidor segons les Figures 4.5 i 4.6 respectivament. Client i servidor estableixen contacte i es fa la transferència inicial, on els bitcoins es dipositen en una

adreça que requereix que signin ambdós per extreure'ls. Per a simular una caiguda del servidor, parem el procés del servidor.

```
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$ ./client.py init -port 9999
-file jsons/paramClient -password uoc2016 -bitcoin 0.05979999 -micropayment 0.01
 -duracio 0.10 -fee 0.0002 -fileStart jsons/startEncrypt -passwordStart bonanyno
u2016 -broadcast 1
Resposta broadcast: {"txid":"9c3cd7c317019ac61f45b9aa8b5ef2929040c39de8a969670cd
8d9a700345654"}
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$ ./client.py newTx -port 9999
 -file jsons/paramClient -password uoc2016
Traceback (most recent call last):
  File "./client.py", line 168, in <module>
    s.connect((host, port))
ConnectionRefusedError: [Errno 111] Connection refused
```

Figura 4.5: Client que ha perdut contacte amb el servidor

```
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$ ./server.py -port 9999 -objec
tiu 0.05 -duracio 0.10 -password misticuoc -broadcast 1
^CTraceback (most recent call last):
  File "./server.py", line 73, in <module>
    conn, addr = s.accept()
  File "/usr/lib/python3.4/socket.py", line 187, in accept
    fd, addr = self._accept()
KeyboardInterrupt
josep@ubuntu:~/Escriptori/python-bitcoinlib-master$
```

Figura 4.6: Servidor caigut

Observem ara que el client intenta realitzar un micropagament però aquest és rebutjat ja que el servidor està caigut. Per tant, en aquests moments el client no sap que succeeix i vol recuperar els bitcoins dipositats a l'adreça comuna. Per fer-ho, usarà la transacció de seguretat que li ha signat inicialment el servidor. Observem a la Figura 4.7 que la transacció és rebutjada en un primer intent d'enviar la transacció. El motiu és que el temps de bloqueig de la transacció que han establit client i servidor encara no ha passat i per tant, el client no pot recuperar per ara els bitcoins.

```
  File "/usr/lib/python3.4/urllib/request.py", line 587, in http_error_default
    raise HTTPError(req.full_url, code, msg, hdrs, fp)
urllib.error.HTTPError: HTTP Error 400: Bad Request
```

Figura 4.7: Broadcast rebutjat

Un cop passat el temps establert (en aquesta prova s'ha decidit usar un temps de 6 minuts per no allargar l'espera), el client intenta realitzar un broadcast de la transacció de seguretat i aquest cop ha funcionat correctament, com es pot veure a la Figura 4.8

Aleshores, el client ha pogut recuperar els diners en cas d'una caiguda del servidor, sempre esperant el temps acordat entre ambdós per a poder fer-ho. A la Figura 4.9, capturada de la blockchain, podem observar que la transacció té definit correctament el paràmetre nLockTime que ens permet establir aquest temps d'espera.

En versions anteriors del protocol Bitcoin hauríem pogut realitzar el broadcast de la transacció sense esperar el temps acordat, però no s'hauria afegit a la blockchain fins a esgotar el temps d'espera. Actualment, com hem pogut comprovar, s'ha d'esperar a que el temps d'espera s'esgoti per a poder realitzar el broadcast.

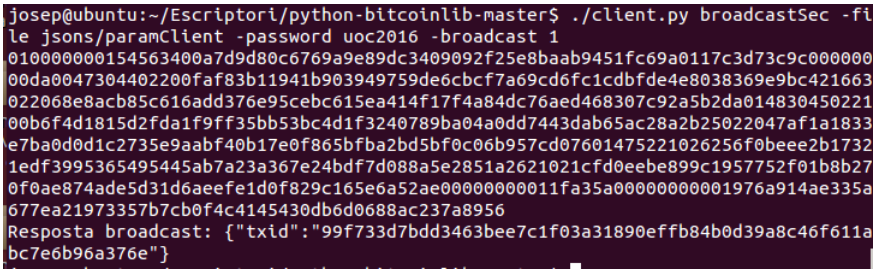

Figura 4.8: Broadcast efectuat correctament

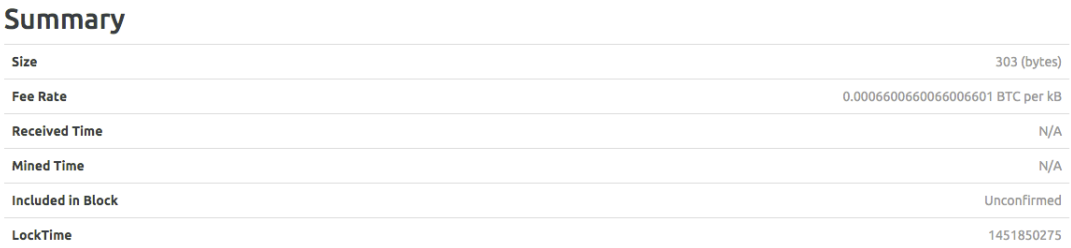

Figura 4.9: Dades de la transacció de seguretat

CAPÍTOL 5

Conclusions

En aquest capítol veurem les conclusions finals del projecte, juntament amb els objectius complerts i els futures línies de treball.

5.1 Conclusions finals

Aquest treball final de màster pretén ser una introducció als *Smart Contracts* i donar les eines necessàries per poder-ne elaborar un en Bitcoin. Com a exemple de contracte intel·ligent, s'ha implementat un canal de micropagaments i s'han realitzat diverses proves amb aquest canal sobre la testnet.

A la introducció als contractes intel·ligents s'han explicat exemples com el DRM, l'intercanvi d'arxius P2P i s'han vist avantatges i inconvenients d'aquests contractes. Després d'una lectura global del capítol, queda clar que el terme *smart contract* és molt genèric i ningú ha estat capaç de definir-lo totalment. Per tant és un àmbit encara molt desconegut per la majoria de gent on encara tot està pràcticament per fer.

Seguidament de la introducció s'han donat les nocions bàsiques de les transaccions en Bitcoin i s'ha explicat el funcionament del llenguatge scripting utilitzat. Les transaccions són complexes d'entendre, ja que tenen molts detalls que fàcilment s'escapen al llegir informació. Si a més tenim en compte la manca d'actualització de la documentació de Bitcoin, encara es complica més aquest estudi.

Pel que fa el llenguatge scripting, s'ha vist que té molt potencial i com s'utilitza en les transaccions bitcoin. Tot i així, també s'arriba a la conclusió que aquest potencial no s'exprimeix degut a les limitacions que es posen en els tipus de transaccions acceptats pels nodes.

Per últim s'ha mostrat que tot i les limitacions del tipus de transaccions de bitcoin és possible realitzar un contracte intel·ligent, agafant d'exemple el canal de micropagaments. Gràcies a les llibreries de Python, la implementació és relativament senzilla i el codi és llegible amb uns mínims coneixements sobre la criptomoneda.

5.2 Objectius complerts

A l'inici del projecte es plantejaren 4 objectius bàsics que s'han assolit durant l'execució d'aquest treball. No ha estat necessari modificar la línia temporal de treball, ja que els terminis s'han ajustat correctament als establerts.

Observem ara al detall com s'han complerts els objectius del projecte, corresponents als objectius de la introducció:

1. Amb la introducció, hem pogut veure el concepte d'smart contract i diferents exemples per entendre'n els conceptes més bàsics.
2. Gràcies a la implementació realitzada en Python, s'ha vist com usar la testnet a nivell d'implementació per a realitzar transaccions.
3. S'ha fet un petit estudi del llenguatge scripting que ofereix bitcoin. A més, s'ha vist quins scripts són vàlids per a les transaccions ja que aquests estan restringits a tipus molt concrets.
4. Finalment s'ha pogut implementar un canal de micropagaments com a exemple de contracte intel·ligent, totalment funcional i que exemplifica el potencial que pot tenir Bitcoin en aquest terreny.

5.3 Futures línies de treball

El treball ha començat explicant els contractes intel·ligents com un concepte molt genèric i a acabat concretant com crear-ne un usant les eines que ofereix el Bitcoin. En tots els punts d'aquest procés, hi ha possibles futures línies de treball i investigació:

- Analitzar amb detall algun altre tipus de contracte intel·ligent, com pot ser el DRM o pensar noves formes de generar un nou tipus de contracte intel·ligent.
- Entendre i concretar l'ús d'oracles en contractes intel·ligents, ja que aquests poden ser la base de futurs nous tipus de contracte.
- Analitzar altres opcions de blockchain diferents de bitcoin, com poden ser BitcoinJ o Etherum. Aquestes opcions estan més orientades a la creació de contractes intel·ligents, tot i que no encara no són tan conegudes.
- Crear nous protocols per a realitzar contractes usant la blockchain de Bitcoin.

- Durant l'execució d'aquest treball, s'ha incorporat a l'última versió del protocol una nova instrucció del llenguatge scripting anomenada `OP_CHECKLOCKTIMEVERIFY`. Aquesta instrucció es presenta com una millora i una facilitat per a realitzar nous contractes intel·ligents i millorar els actuals. Un estudi del que fa aquesta instrucció i com milloraria el protocol presentat pot ser una línia de treball amb molt potencial.

APÈNDIX A

Anàlisi del codi implementat

La finalitat d'aquest apèndix és mostrar el codi implementat per a l'elaboració del canal de micropagament explicat al Capítol 4. Per fer-ho, s'obviaran detalls del codi com la lectura de paràmetres o l'encriptació AES i és farà èmfasi a les funcions principals del client, el servidor i les funcions comunes que utilitzen.

A.1 El client

Recordem que el client, a diferència del servidor, no està permanent en execució. Les accions que realitza el client es passen per paràmetre al moment d'executar-lo. Per això el client analitza el primer argument de la crida i és classifica així:

```
if sys.argv[1] == 'init':
    ...
elif sys.argv[1] == 'newTx':
    ...
elif sys.argv[1] == 'broadcastSec':
    ...
```

Veiem ara la inicialització del client. La primera part consta de la creació d'un diccionari on guardem tota la informació necessària per a executar el protocol i de la lectura de la clau privada de l'adreça inicial per a realitzar la primera transacció.

```
info = {'secKeyClient': '',
            'pubKeyClient': '',
            'pubKeyServer': '',
            'addressClient': '',
```

```
            'addressServer': '',
            'addressCommon': '',
            'secTx': '',
            'secTxId': '',
            'initTx': '',
            'initTxId': '',
            'it': 0,
            'btc': btc - fee,
            'mpy': mpy
            }

    # Llegim params inicials encriptats
    cipherStart = AESCipher(key=passwordStart)

    with codecs.open(fileNameStart, 'r', 'utf-8') as infile:
        params = eval(cipherStart.decrypt(infile.read()))
        infile.close()

    wapSecKeyStart = params['wapSecKeyStart']
    txIdStart = params['txIdStart']
    txVoutStart = int(params['txVoutStart'])

    # Incorporem la clau privada de l'address de sortida
    secKeyIni = CBitcoinSecret(wapSecKeyStart)
    addressIni = get_address(secKeyIni.pub)
```

El següent pas és generar un parell de claus aleatòries per a intercanviar-les amb el servidor i així obtenir l'adreça comuna on dipositarem els bitcoins.

```
# Generem una clau aleatoria  publica
    secKeyClient = CBitcoinSecret.from_secret_bytes(sha256(os.urandom(32)).digest())
    info['secKeyClient'] = str(secKeyClient)
    info['pubKeyClient'] = b2x(secKeyClient.pub)
    addressClient = get_address(secKeyClient.pub)
    info['addressClient'] = str(addressClient)

    # Enviem clau publica al servidor i obtenim la seva
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
    s.sendall(b'newClient')
    if s.recv(4096) != b'ok':
        print('Error de connexio')
        quit()
    s.sendall(secKeyClient.pub)
    pubKeyServer = s.recv(4096)
    info['pubKeyServer'] = b2x(pubKeyServer)
```

```
# Generem address server
addressServer = get_address(pubKeyServer)
info['addressServer'] = str(addressServer)

# Generem address de pagament comuna
addressCommon = get_address_multisign(secKeyClient.pub, pubKeyServer)
info['addressCommon'] = str(addressCommon)
```

Un cop realitzat l'intercanvi de claus i calculat el necessari, es procedeix a generar la transacció inicial i signar-la per a posteriorment enviar-ho al servidor.

```
# Generem la transaccio inicial
txIni = new_tx(txIdStart, txVoutStart, [(str(addressCommon), btc - fee)])

# Signem la transaccio inicial
txIni.vin[0].scriptSig = sign_tx_pkh(secKeyIni, txIni)
info['initTx'] = b2x(txIni.serialize())
txIniId = get_hash_tx(txIni)
info['initTxId'] = txIniId

# Generem Tx de seguretat
txSec = new_tx(txIniId, 0, [(str(addressClient), btc - 2 * fee)],
                            int(time.time()) + int(60 * 60 * timeSec))

# Enviem Tx al Servidor i signem
s.sendall(txSec.serialize())
sigSecServer = s.recv(4096)
sigSecClient, script = sign_tx_psh(secKeyClient, secKeyClient.pub, pubKeyServer, txSec)
txSec.vin[0].scriptSig = CScript([bytes(0), sigSecClient, sigSecServer, script])
info['secTx'] = b2x(txSec.serialize())
txSecId = get_hash_tx(txSec)
info['secTxId'] = txSecId
s.close()
```

Per últim guardem tota la informació en un fitxer JSON encriptat amb l'AES.

```
# Guardem diccionari a un fitxer JSON encriptat
with codecs.open(fileName, 'w', 'utf-8') as outfile:
    outfile.write(cipher.encrypt(json.dumps(info)))
    outfile.close()

broadcast_tx(b2x(txIni.serialize()), optionBroadcast)
```

Passem ara al codi de generar una nova transacció. Aquest consta de tres parts diferenciades: recuperació de la informació guardada en la inicialització del client, creació d'una nova transacció juntament amb l'enviament al servidor i guardar de nou les dades al fitxer JSON encriptat. La

primera i l'última part són similars al codi de la inicialització. Per altra banda, el codi de la nova transacció és el següent:

```
# Recuperem el diccionari amb la informacio
with codecs.open(fileName, 'r', 'utf-8') as infile:
    info = eval(cipher.decrypt(infile.read()))
    infile.close()

info['it'] += 1

# Generem Tx i
txi = new_tx(info['initTxId'], 0, [(info['addressClient'],
                                    info['btc'] - info['it'] * info['mpy'] - fee),
                                   (info['addressServer'], info['it'] * info['mpy'])])

# Enviem Tx i la seva signatura al servidor
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.sendall(b'newTxI')
if s.recv(4096) != b'ok':
    print('Error de connexio')
    quit()
secKeyClient = CBitcoinSecret(info['secKeyClient'])
s.sendall(secKeyClient.pub)
if s.recv(4096) != b'ok':
    print('Error de connexio')
    quit()
s.sendall(txi.serialize())
if s.recv(4096) != b'ok':
    print('Error de connexio')
    quit()
sigTxiClient, script = sign_tx_psh(secKeyClient, secKeyClient.pub,
                                   x(info['pubKeyServer']), txi)
s.sendall(sigTxiClient)

# Guardem diccionari a un fitxer JSON encriptat
with codecs.open(fileName, 'w', 'utf-8') as outfile:
    outfile.write(cipher.encrypt(json.dumps(info)))
    outfile.close()
```

Per últim, l'acció de realitzar broadcast de la transacció de seguretat és molt senzilla. Només cal obrir el fitxer encriptat JSON, buscar-ne la transacció de seguretat i realitzar-ne broadcast.

```
with codecs.open(fileName, 'r', 'utf-8') as infile:
    info = eval(cipher.decrypt(infile.read()))
    infile.close()
```

```
    print(info['secTx'])
    broadcast_tx(str(info['secTx']), optionBroadcast)
```

A.2 El servidor

Recordem que el servidor, a diferència del client, està permanentment en execució fins que o bé s'esgota el temps especificat o bé s'arriba a un objectiu de pagaments. El control d'aquest esquema es fa amb un bucle infinit que permanentment escolta a un socket. En cas d'esgotar el temps de duració del socket, es captura l'excepció generada per a realitzar el broadcast de les transaccions acumulades. El mateix succeeix en cas que s'arribi a l'objectiu especificat.

```
while True:
    # En cas d'arribar a l'objetiu fixat, realitzem broadcast
    # de les transaccions aconseguides
    if btcTotal >= btcObjectiu:
        print ('S ha arribat a l objectiu. Broadcast de totes les transaccions.')
        for fileName in os.listdir('jsons/server/'):
            with codecs.open('jsons/server/' + fileName, 'r', 'utf-8') as infile:
                info = eval(cipher.decrypt(infile.read()))
                infile.close()
            print(info['txs'][info['it'] - 1]['tx'])
            broadcast_tx(str(info['txs'][info['it'] - 1]['tx']), optionBroadcast)
        quit()

    # Intentem rebre una peticio al socket. En cas d'excedir
    # la duracio del canal, es fa broadcast de les transaccions
    try:
        # Esperem l'entrada d'una peticio
        conn, addr = s.accept()
        instruccio = conn.recv(4096)

        # Cas que entri un nou client
        if instruccio == b'newClient':
            ...

        # Cas que un client existent ens entregui una nova
        # transaccio
        elif instruccio == b'newTxI':
            ...

        # En cas de rebre una altra instruccio, informem
        # per pantalla
        else:
            print('Instruccio desconeguda')
```

```
# Tractament de l'excepcio quan s'esgota la duracio del socket
except socket.error as e:
    ...
```

El servidor, a part de realitzar el broadcast de les transaccions acumulades només realitza dues accions: inicialitzar el protocol amb un nou client o rebre nous pagaments d'un client ja existent. En el primer cas el codi és molt semblant al client, ja que la inicialització és la mateixa. Difereix pel fet que no crea cap transacció, sinó que només en signa una i retorna aquesta signatura.

```
        # Generem un diccionari on guardar la informacio
        # del client
        info = {'secKeyServer': '',
                'pubKeyServer': '',
                'pubKeyClient': '',
                'addressClient': '',
                'addressServer': '',
                'addressCommon': '',
                'txs': [],
                'it': 0
                }

        conn.sendall(b'ok')

        # Rebem la clau publica del client
        pubKeyClient = conn.recv(4096)
        info['pubKeyClient'] = b2x(pubKeyClient)
        addressClient = get_address(pubKeyClient)
        info['addressClient'] = str(addressClient)

        # Generem claus del servidor
        secKeyServer = CBitcoinSecret.from_secret_bytes(sha256(os.urandom(32)).digest())
        conn.sendall(secKeyServer.pub)
        info['secKeyServer'] = str(secKeyServer)
        info['pubKeyServer'] = b2x(secKeyServer.pub)

        # Generem address del servidor
        addressServer = get_address(secKeyServer.pub)
        info['addressServer'] = str(addressServer)

        # Generem address de pagament comuna
        addressCommon = get_address_multisign(pubKeyClient, secKeyServer.pub)
        info['addressCommon'] = str(addressCommon)

        # Rebem i signem tx de seguretat
        data = conn.recv(4096)
```

```
        txSeguretat = CMutableTransaction.deserialize(data)
        sign, script = sign_tx_psh(secKeyServer, pubKeyClient,
                                   secKeyServer.pub, txSeguretat)
        conn.sendall(sign)
        conn.close()

        # Guardem la transaccio en un fitxer JSON encriptat
        with codecs.open('jsons/server/' + b2x(pubKeyClient), 'w', 'utf-8') as outfile:
            outfile.write(cipher.encrypt(json.dumps(info)))
            outfile.close()
```

Quan rep un nou pagament d'un client, el servidor senzillament signa la transacció i la guarda juntament amb les altres dades necessàries per a identificar el client.

```
        conn.sendall(b'ok')
        pubKeyClient = conn.recv(4096)
        conn.sendall(b'ok')

        # Recuperem el diccionari amb la informacio
        with codecs.open('jsons/server/' + b2x(pubKeyClient), 'r', 'utf-8') as infile:
            info = eval(cipher.decrypt(infile.read()))
            infile.close()

        info['it'] += 1

        # Rebem la nova transaccio
        data = conn.recv(4096)
        txi = CMutableTransaction.from_tx(CMutableTransaction.deserialize(data))
        conn.sendall(b'ok')

        # Rebem la signatura del client
        signClient = conn.recv(4096)
        conn.close()

        # Signem la transaccio
        secKeyServer = CBitcoinSecret(info['secKeyServer'])
        signServer, script = sign_tx_psh(secKeyServer, pubKeyClient, secKeyServer.pub, txi)
        txi.vin[0].scriptSig = CScript([bytes(0), signClient, signServer, script])
        txId = get_hash_tx(txi)

        # Afegim la transaccio signada al diccionari
        info['txs'].append({'it': info['it'], 'btc': 1.0 * txi.vout[1].nValue / COIN,
                            'txId': txId, 'tx': b2x(txi.serialize())})

        if info['it'] > 1:
            btcTotal -= 1.0 * info['txs'][info['it'] - 2]['btc']
```

```
            btcTotal += 1.0 * txi.vout[1].nValue / COIN

            print('Nou donatiu. BTC actual:', btcTotal)

            # Guardem diccionari a un fitxer JSON encriptat
            with codecs.open('jsons/server/' + b2x(pubKeyClient), 'w', 'utf-8') as outfile:
                outfile.write(cipher.encrypt(json.dumps(info)))
                outfile.close()
            b2x(pubKeyClient)
```

A.3 Funcions comunes

Per a la implementació del client i servidor s'ha usat la llibreria bitcoinlib, la qual proporciona les eines necessàries per a gestionar les transaccions bitcoin. Tot i així, el seu ús és bastant tediós i poc intuïtiu. Per a simplificar el codi, s'ha creat un arxiu anomenat funcions.py el qual es usat tant pel client com pel servidor. Aquest arxiu inclou algunes funcions que es repeteixen diverses vegades al codi i que a més, estan adaptades al protocol de micropagaments.

La primera funció que s'inclou és la creació d'una nova transacció. Per a simplificar, es suposa que són transaccions que provenen d'una sola entrada i se li poden incloure les sortides que es vulguin, usant tuples (adreça, quantitat). A més, ofereix l'opció d'afegir un paràmetre nLockTime per a bloquejar temporalment la transacció. L'ús d'aquest paràmetre, requereix que el paràmetre nSequence s'inicialitzi a 0.

```
def new_tx(txIdIn, vout, outs, nLockTime=0):
    txin = CMutableTxIn(COutPoint(lx(txIdIn), vout))
    if nLockTime > 0:
        txin.nSequence = 0
    txins = [txin]
    txouts = []
    for out in outs:
        txouts.append(CMutableTxOut(out[1] * COIN, CBitcoinAddress(out[0]).to_scriptPubKey()))
    tx = CMutableTransaction(txins, txouts)
    tx.nLockTime = nLockTime
    return tx
```

Les següents funcions creades serveixen per signar diferents tipus de transaccions: les PKH i les PSH. Els retorns de les funcions són diferents, ja que per a incorporar les signatures a les transaccions es necessiten paràmetres diferents segon el tipus de transacció.

```
def sign_tx_pkh(seckey, tx):
    txin_scriptPubKey = CScript([OP_DUP, OP_HASH160, Hash160(seckey.pub),
                                 OP_EQUALVERIFY, OP_CHECKSIG])
    sighash = SignatureHash(txin_scriptPubKey, tx, 0, SIGHASH_ALL)
    sig = seckey.sign(sighash) + bytes([SIGHASH_ALL])
```

```
    return CScript([sig, seckey.pub])



def sign_tx_psh(seckey, pubkeyUser, pubkeyServer, tx):
    txin_redeemScript = CScript([OP_2, pubkeyUser, pubkeyServer, OP_2, OP_CHECKMULTISIG])
    sighash = SignatureHash(txin_redeemScript, tx, 0, SIGHASH_ALL)
    return seckey.sign(sighash) + bytes([SIGHASH_ALL]), txin_redeemScript
```

Seguidament es mostren tres funcions molt senzilles que ens permeten obtenir fàcilment les adreces a partir de les claus públiques i la id d'una transacció:

```
def get_address(pubKey):
    return P2PKHBitcoinAddress.from_pubkey(pubKey)



def get_address_multisign(pubKeyClient, pubKeyServer):
    txin_redeemScript = CScript([OP_2, pubKeyClient, pubKeyServer, OP_2, OP_CHECKMULTISIG])
    txin_scriptPubKey = txin_redeemScript.to_p2sh_scriptPubKey()
    return CBitcoinAddress.from_scriptPubKey(txin_scriptPubKey)



def get_hash_tx(tx):
    return b2lx(sha256(sha256(tx.serialize()).digest()).digest())
```

Per últim s'ha implementar la funció per a realitzar broadcast d'una transacció. Preveient que un node puntualment pot fallar i no acceptar transaccions, es dóna l'opció d'usar dos nodes diferents per al broadcast. Les APIs que proporcionen utilitzen un JSON diferent com a paràmetre, la qual cosa es tractada al codi:

```
def broadcast_tx(tx, optionBroadcast):
    if optionBroadcast == 1:
        url = 'https://testnet.blockexplorer.com/api/tx/send'
        values = {'rawtx': tx}
    else:
        url = 'http://tbtc.blockr.io/api/v1/tx/push'
        values = {'hex': tx}
    params = json.dumps(values).encode('utf8')
    req = urllib.request.Request(url, data=params,
                                 headers={'content-type': 'application/json'})
    response = urllib.request.urlopen(req)
    print('Resposta broadcast:', response.read().decode('utf8'))
```

Bibliografia

[1] Transactions. https://bitcoin.org/en/developer-guide#transactions, 2009-2015.

[2] Script. https://en.bitcoin.it/wiki/Script, 2010.

[3] Contract. https://en.bitcoin.it/wiki/Contract, 2011.

[4] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1a edition, 2014.

[5] Manuel Aráoz. Smart contracts and bitcoin. https://medium.com/@maraoz/smart-contracts-and-bitcoin-a5d61011d9b1#.rz664ay0i, 2015.

[6] Florian Glatz. What are smart contracts? https://medium.com/@heckerhut/whats-a-smart-contract-in-search-of-a-consensus-c268c830a8ad#.tix02yrpz, 2014.

[7] Nathaniel Karp. Tecnología de cadena de bloques (blockchain): la última disrupción en el sistema financiero. https://www.bbvaresearch.com/wp-content/uploads/2015/07/150714_US_EW_BlockchainTechnology_esp.pdf, 2015.

[8] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *The Cryptography Mailing List*, 2008. Disponible a: http://bitcoin.org/bitcoin.pdf . Última consulta 04-01-2015.

[9] David Tuesta. Smart contracts: ¿lo último en automatización de la confianza? https://www.bbvaresearch.com/wp-content/uploads/2015/10/Situacion_Ec_Digital_Oct15_Cap1.pdf, 2015.

Josep Miquel Andreu
Barcelona, 2016