Università di Pisa

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Informatica

Progettazione e implementazione di una DHT sicura integrata con servizi Cloud

Candidato:
Daniele Antuzi

Relatore:
Prof.ssa Laura Ricci

Sessione di Laurea 5 Dicembre 2014
Anno Accademico 2013/2014

Indice

Introduzione

Negli ultimi anni, in seguito all'espansione dei social network e di servizi cloud quali Dropbox o Google Drive, gli utenti della rete sono sempre più incoraggiati a memorizzare i propri dati su piattaforme online. Per esempio, grazie al mantenimento dei propri contenuti su un cloud remoto, gli utenti hanno la possibilità di accedere ad essi da qualunque dispositivo connesso ad internet. Inoltre, in caso di guasto improvviso del proprio personal computer, possono disporre di informazioni di backup memorizzate su uno di questi servizi di cloud storage.

Purtroppo questo implica una perdita di privacy sui dati e, se si considerano gli avvenimenti e gli scandali degli ultimi anni (PRISM, DATAGATE), ci si rende conto di come questi colossi mondiali possano tracciare le nostre vite. I metadati riguardanti le conversazioni e le liste di amici degli utenti dei social network possono venire incrociati per individuare i rapporti tra le persone, ottenendo delle analisi che potrebbero essere utili, per esempio, per prevenire atti di terrorismo. Analizzando uno scenario più semplice, un hacker può riuscire ad intromettersi in qualche sistema di storage per rubare informazioni riservate. Risulta quindi evidente la necessità di definire sistemi che garantiscano un maggior livello di sicurezza.

Il problema della sicurezza dei nostri dati e delle nostre comunicazioni è sempre più attuale, ne è prova la recente diffusione di applicazioni cifrate di messaggistica o di invio file. Molti di questi servizi però utilizzano solo una parte delle tecniche che possono rendere il sistema sicuro. Analizzando, ad esempio, alcune applicazioni di messaggistica o di storage con programmi come Burp

Suite[27] con funzione di proxy e programmi come Wireshark[18] utilizzati per "sniffare"[1] i pacchetti sulla rete, si deduce che la maggior parte delle applicazioni si limitano ad utilizzare il protocollo SSL per la comunicazione tra client e server. Questo significa che i dati viaggiano cifrati sulla rete però sul server vengono decifrati e memorizzati in chiaro.

Questa soluzione garantisce che un potenziale hacker non possa leggere i dati che sono trasmessi sulla rete, però se esso riesce ad individuare una vulnerabilità e accedere al database del server può avere a disposizione una quantità di informazioni non indifferente. Altre applicazioni aggiungono al protocollo SSL una cifratura sul server con chiavi generate sul server stesso. Nonostante una soluzione del genere offra un livello di protezione migliore, non garantisce che i gestori del servizio non abbiano la possibilità di accedere a contenuti riservati come, ad esempio, i nostri file o la cronologia delle nostre conversazioni.

Presso l'azienda eLearnSecurity srl[14] di Pisa, il problema della riservatezza dei dati su cloud si sta affrontando già da qualche anno. Per risolvere parzialmente il problema è stato sviluppato un servizio, Justcrypt[36], che permette di inviare file in maniera del tutto sicura con cifratura lato client. Su Justcrypt, il file viene cifrato con una chiave casuale generata sul client web del mittente, caricato su un server e scaricato dal destinatario attraverso un una URL. Questo sistema tuttavia presenta delle limitazioni in quanto la chiave per la decifratura del file deve essere inviata al destinatario attraverso un servizio esterno indipendente da Justcrypt.

Proprio durante lo sviluppo di Justcrypt è nata quindi l'idea, sempre presso eLearnSecurity, della progettazione di un sistema per gestire in maniera del tutto sicura un vasto range di servizi come, ad esempio, messaggistica, invio di file, storage distribuito, ecc.

L'obiettivo di questa tesi è stato quello di progettare un sistema con queste caratteristiche. La tesi presenta Izzie, un'infrastruttura che permette la creazione di servizi estendibili e personalizzabili che sfruttano l'integrazione delle

[1] Il termine sniffare è utilizzato per indicare l'azione mediante la quale si intercettano e si leggono tutti i pacchetti dati che passano su una parte della rete locale

tecnologie cloud e P2P per offrire all'utente finale dei benefici in termini di affidabilità, efficienza, costi e soprattutto sicurezza.

L'architettura di Izzie si può suddividere in tre blocchi principali: **client**, **autenticazione** e **controllo**. Per ogni servizio che si vuole implementare sfruttando l'architettura Izzie è necessario definire un controllo del servizio stesso. Per molte applicazioni, inoltre, è richiesta l'autenticazione del cliente che si può ottenere grazie ad un servizio appartenente al blocco di autenticazione. Fanno potenzialmente parte di questo blocco tutti i servizi che implementano il protocollo OAuth 2.0[20] come Facebook, Google, Twitter, LinkedIn, ecc. Per richiede l'autenticazione degli utenti, il controllo di un servizio deve implementare la parte del protocollo OAuth necessaria all'integrazione con uno o più servizi di autenticazione. Il client offre un'interfaccia utente unica per gestire più servizi e riesce a comunicare con istanze diverse del blocco di autenticazione grazie al protocollo OAuth descritto nella RFC 6759[20] mentre, la comunicazione con istanze diverse del blocco controllo è garantita dalla presenza di una API REST[16] che deve essere la stessa per tutti i controlli.

Grazie alla suddivisione dell'architettura in blocchi distinti si riesce a semplificare la personalizzazione del sistema. Consideriamo il caso di una azienda che voglia sviluppare un servizio di chat aziendale. Si può decidere di implementare un proprio controllo della chat ed eseguirlo su un server interno mentre, come autenticazione, si potrebbe prevedere che l'utente utilizzi il proprio account Google. L'azienda, inoltre, modificando opportunamente il controllo, potrebbe prevedere di salvare la cronologia delle conversazioni su un servizio di cloud storage. Nonostante le diverse personalizzazioni che si possono effettuare su controllo e autenticazione, una volta configurato opportunamente il client per interagire con i servizi di sviluppati, un dipendente dell'azienda, attraverso il proprio client già installato, avrà accesso al nuovo servizio di chat aziendale.

Izzie prevede che il client possa implementare un protocollo distribuito che permetta di definire una rete P2P che collega i client. I servizi cloud offorno un insieme di risorse di calcolo e/o storage con una tariffazione dipendente esclusivamente dalla reale quantità di risorse utilizzate garantendo alta affida-

bilità. La tecnologia P2P, d'alto canto, si basa sulla condivisione di una parte delle risorse private al fine di creare un sistema completamente distribuito che possa offrire, gratuitamente, servizi agli utenti della rete.

Un servizio progettato su una rete P2P è completamente gratuito e potrebbe essere la scelta ideale per quelle applicazioni che non risultano particolarmente sensibili alle latenze necessarie per il reperimento di un contenuto. Esistono inoltre delle applicazioni che sono per loro natura distribuite. Basti pensare ad un servizio di messaggistica, file sharing, ecc. Per questi tipi di applicazioni un sistema distribuito è l'ideale per raggiungere delle basse latenze di comunicazione e ridurre la probabilità di rischio di intercettazione.

Il sistema Izzie è stato progettato come un sistema molto generale, quindi deve poter supportare sia applicazioni che richiedono uno storage affidabile su cloud, sia altre applicazioni che richiedono uno storage gratuito implementato da una Distributed Hash Table(DHT)[37]. Una DHT è un sistema di storage distribuito nel quale ogni informazione è associata ad una chiave e, in base al valore di quest'ultima, il dato viene memorizzato su un determinato insieme di peer.

Ipotizziamo, ad esempio, che il gestore di un quotidiano voglia offrire la possibilità agli abbonati di scaricare tutti i giornali pubblicati nel periodo compreso nell'abbonamento. Si avrà la necessità di mantenere un archivio di tutte le edizioni pubblicate su un servizio di cloud storage in modo che un guasto all'interno dei datacenter che ospitano il servizio non comprometta la raggiungibilità dei dati. Uno dei maggiori costi che bisogna sostenere per mantenere un'applicazione di questo tipo, consiste nella banda di upload del server cloud. Per ogni edizione che un utente vuole scaricare, la redazione deve sostenere dei costi per la banda di upload che, presi singolarmente sono irrisori ma accumulati su tutti gli utenti diventano considerevoli. Si consideri che ogni utente, una volta autenticato, avrà la possibilità di scaricare tutte le edizioni che vuole ma, molto probabilmente, scaricherà solamente l'edizione del giorno stesso. Sulla rete ci sarà quindi, con alta probabilità, un elevato numero di persone che possiedono una copia del quotidiano e che, quindi,

possono condividerla attraverso la DHT con chi, possedendo i diritti, ne farà richiesta. Grazie a questo servizio di storage ibrido cloud/P2P offerto da Izzie, la redazione del giornale potrebbe ottenere un notevole risparmio derivante dalla presenza di una DHT utilizzata come cache distribuita tra i client degli utenti.

Oltre all'elevato numero di personalizzazioni possibili e all'elevato livello di usabilità, uno dei punti di forza dell'architettura Izzie è la sicurezza. Si prevede infatti una gestione dell'architettura e delle chiavi di cifratura per garantire la riservatezza sui servizi cloud mentre, nel caso del P2P, è stata progettata una DHT che preveda un elevato livello di sicurezza.

La maggior parte delle DHT nasce per come middleware di applicazioni di filesharing come eMule[29], Kazaa, BitTorrent[23], ecc. In questi sistemi i contenuti memorizzati nella DHT sono condivisi, quindi, non si presenta la necessità di progettare sistemi di sicurezza che permettano di assegnare dei diritti in lettura o scrittura ai dati memorizzati sulla rete. Per questo motivo, nelle reti P2P che conosciamo e utilizziamo, tutti gli utilizzatori sono anonimi e hanno i diritti su tutti i dati che possono transitare sulla rete. Come analizzeremo in dettaglio nel capitolo 2, le attuali DHT sono soggette ad un numero elevato di attacchi che un malintenzionato può effettuare.

In questa tesi abbiamo progettato un sistema P2P che prevenga la totalità degli attacchi più comuni a cui è soggetta una DHT. Tutti i dati memorizzati nella DHT sono cifrati sia quando risiedono su un nodo che quando vengono trasmessi ad altri nodi. Per garantire una maggiore sicurezza, alcuni servizi possono richiedere che alcuni dati siano salvati solamente tra i dispositivi utilizzati da utenti appartenenti ad un determinato gruppo e che nessun altro peer della rete debba entrare in possesso di tali informazioni nonostante queste siano cifrate.

Ipotizziamo che un'azienda voglia mantenere dei documenti riservati mediante un servizio di storage distribuito che voglia utilizzare solo i computer interni e quelli degli impiegati. Il sistema di storage deve poter salvare i file sui peer utilizzati dagli utenti appartenenti ad un determinato gruppo creando

così una nuova DHT i cui nodi sono un sottoinsieme di quella generale. Ipotizziamo ora che la stessa azienda voglia salvare dei contenuti solo sui computer utilizzati dal gruppo di dirigenti. Questo significa che i client utilizzati da questi dirigenti dovranno poter gestire documenti riservati ai dirigenti, documenti aziendali e file memorizzati sulla rete globale. Si passa così dal concetto di DHT "flat" a un'organizzazione gerarchica formata da gruppi che possono annidarsi e intersecarsi.

Il progetto dell'architettura Izzie è suddiviso in due parti. In questa tesi ci si è concentrati sulla parte P2P progettando un protocollo che prevenga gli attacchi noti ai quali è soggetto un sistema distribuito. Il blocco controllo e autenticazione sono stati sviluppati in un'altra tesi pertanto la loro descrizione non verrà approfondita. Il contributo dato in questo lavoro riguarda principalmente i seguenti punti.

- **progettazione** dell'architettura complessiva di Izzie;
- **studio** delle vulnerabilità delle attuali DHT;
- **analisi** delle soluzioni di sicurezza proposte in letteratura;
- **progettazione** del protocollo relativo alla DHT sicura;
- **implementazione** di una libreria per la comunicazione diretta e tra peer con cifratura end-to-end e integrata con i servizi Izzie.

In particolare, la tesi propone molte innovazioni per la progettazione della DHT tra cui le più importanti sono:

- **tabella di routing tridimensionale** per permettere ad un peer, durante la ricerca di una chiave, di analizzare più percorsi paralleli. Questa tecnica è resa necessaria perché viene considerata l'ipotesi secondo la quale qualche peer vicino sia compromesso da un utente malevolo.
- gestione di un **valore di affidabilità** per giudicare il comportamento di altri peer escludendoli dalla rete se si ritiene che questi appartenano a utenti malevoli;

- introduzione e gestione di gruppi di peer per aumentare la sicurezza e affidabilità dei dati appartenenti ad aziende riducendo i costi.

Il progetto è ancora in via di sviluppo e per l'implementazione, è stato applicata una strategia bottom-up. Come primo obiettivo è stata sviluppata una libreria che consenta l'integrazione lato client con gli altri blocchi del sistema e la comunicazione sicura tra due peer. I maggiori problemi nello sviluppo di questa libreria sono gestire la presenza dei router NAT che, in mancanza di impostazioni specifiche, impediscono la creazione di una connessione tra due peer. Sono state quindi implementati protocolli come "PortMapping", "connection reversal" e "hole punching" che verranno descritti nei capitoli successivi. Inoltre, una volta stabilita la connessione, si sono utilizzate tecniche per la creazione di una chiave di sessione, per la cifratura e per la firma digitale.

Nei prossimi capitoli analizzeremo inizialmente(cap.1) l'organizzazione delle DHT prendendo come caso di studio la DHT Pastry. Successivamente(cap.2) ci concentreremo sulle vulnerabilità delle reti P2P attuali discutendo i risultati dell'analisi svolta e presenteremo delle tecniche di cifratura(cap.3) utilizzata da Izzie. Nel capitolo successivo sarà descritta l'architettura generale di Izzie(cap.4), in particolare verranno presentati i componenti base: autenticazione, controllo e client. Il client costituisce anche un peer della DHT che verrà presentata nel capitolo successivo (cap.5). Concludendo la trattazione(cap.6), si discuteranno i principali problemi affrontati durante lo sviluppo e si mostreranno le soluzioni implementate. Nel capitolo 7, infine, riporteremo le conclusioni ed illustreremo alcuni sviluppi futuri.

Capitolo 1

La DHT Pastry

I sistemi P2P sono sistemi distribuiti che hanno avuto una notevole espansione grazie a programmi di condivisione di file che, negli ultimi anni, sono sempre più utilizzati. Un sistema distribuito è una tipologia di sistema informatico costituito da un insieme di processi interconnessi tra loro attraverso un continuo scambio di messaggi. Ogni processo è progettato in modo da collaborare con gli altri ed eseguire un algoritmo distribuito al fine di raggiungere un obiettivo comune. Elenchiamo alcune caratteristiche che più ci interessano dei sistemi distribuiti.

L'eterogeneità la possibilità che i processi siano entità completamente diverse. Possono essere scritti in linguaggi di programmazione diversi, utilizzare sistemi operativi e hardware diversi.

L'assenza di un clock globale che comporta l'impossibilità di sincronizzare precisamente i vari processi.

La scalabilità cioè la capacità di offrire le medesime prestazioni, in termini di throughtput e latenza, indipendentemente dal numero di utilizzatori del sistema.

Una rete P2P è un'architettura in cui i nodi non sono gerarchizzati sotto forma di client o server ma sotto forma di nodi equivalenti o paritari (in inglese peer) che possono cioè fungere sia da cliente che da servente verso gli altri host della

rete. Le comunicazioni tra peer avvengono al di sopra di un'overlay network cioè una rete virtuale costruita al di sopra della rete fisica. Sfruttando questa overlay network, i protocolli P2P riescono ad implementare sistemi di storage distribuito, ovvero Distributed Hash Table(DHT) dove l'inserimento e ricerca di informazioni avviene mediante l'utilizzo di una chiave. Una DHT si deve adattare alle modifiche della topologia della rete visto che, soprattutto in un sistema P2P, ci sono tanti nodi che in ogni momento chiedono di entrare a far parte della rete oppure escono dalla rete stessa. Questo fenomeno viene indicato con il nome di churn. Esistono molti protocolli per creare e gestire una DHT come ad esempio Chord[35], CAN[30], Tapestry[44] o Kademlia[28]

Pastry è un protocollo distribuito e scalabile per implementare una DHT. Ogni nodo della rete Pastry, così come ogni dato salvato in essa, è identificato da un ID. Il protocollo si occupa di mappare su uno stesso spazio delle chiavi sia i nodi che i dati memorizzati. In questo modo risulta facile salvare un dato la cui chiave è X: basta memorizzarlo sui k nodi con l'ID numericamente più vicino ad X. Il dato è replicato su k nodi per garantire che il dato stesso non venga perso a causa del fallimento di un singolo nodo. La probabilità di perdere il dato è uguale alla probabilità che i k nodi che lo contengono falliscano tutti nello stesso momento.

Sia N il numero dei peer connessi alla rete, la ricerca di un dato ha bisogno, con alta probabilità, di un numero di messaggi $O(logN)$.

1.1 Le DHT

Le DHT sono una classe di sistemi distribuiti che partizionano un set di chiavi tra i nodi partecipanti al sistema distribuito stesso. Ogni nodo può inserire delle informazioni nel sistema e può ricercare le informazioni inserite da altri nodi attraverso la chiave che identifica l'informazione. Come conseguenza di quanto detto, una DHT deve implementare un'interfaccia con i seguenti metodi:

- $put(K, Info)$: inserisce l'informazione $Info$ identificata dalla chiave K nella rete.

- *get*(K): avvia il protocollo di routing per ottenere l'informazione identificata dalla chiave K

Durante la ricerca, i nodi sono in grado di inoltrare in maniera efficiente i messaggi all'unico proprietario di una determinata chiave. Una DHT deve:

Essere distribuita : Ogni nodo esegue lo stesso algoritmo, quindi non c'è nessun nodo che coordina gli altri.

Essere tollerante alle modifiche della topologia : Il sistema deve prevedere che alcuni nodi possano entrare, uscire dalla DHT oppure possono fallire. In questo caso è necessaria una riorganizzazione della rete.

Essere scalabile : Il sistema deve funzionare efficientemente anche in presenza di milioni di nodi.

Mantenere il bilanciamento dei nodi : A tutti i nodi deve essere assegnato approssimativamente lo stesso numero di chiavi.

Per raggiungere questi scopi, ogni nodo della DHT mantiene dei collegamenti ad altri nodi chiamati "vicini". Inoltre, per essere tolleranti ai guasti, durante l'operazione di *put*, è necessario salvare una chiave su più nodi per evitare di perdere l'informazione dopo il fallimento del nodo che la contiene. L'organizzazione di una DHT deve tener conto di almeno due parametri:

- il numero di messaggi necessari per il salvataggio e la ricerca di una chiave;
- la quantità di memoria occupata da ogni nodo.

Una forte ottimizzazione della prima di queste proprietà comporta un grosso costo da parte della seconda e viceversa. Prendiamo come esempio una rete composta da N nodi, che vuole minimizzare il numero di messaggi scambiati quindi ogni nodo deve mantenere in memoria un link per ogni nodo della rete. In questo caso il routing comporta un solo messaggio perché, avendo conoscenza di tutta la rete, possiamo inviare la richiesta direttamente al nodo che contiene l'informazione da noi cercata. L'aspetto negativo di quest'approccio è che dobbiamo mantenere una tabella di routing con $O(N)$ entrate. Un caso

diametralmente opposto si ha quando, per minimizzare la memoria utilizzata, ciascun nodo mantiene un solo vicino. In questo caso la rete assume una conformazione ad anello e la ricerca di una chiave costa al massimo O(N). Una DHT si pone come obiettivo quello di trovare il giusto compromesso tra il numero di messaggi necessari per il routing e il numero di entrate nella tabella di routing:

- $O(logN)$: messaggi per il routing
- $O(logN)$: entrate nella tabella di routing

Per raggiungere il suddetto tradeoff, una DHT deve mantenere una struttura più o meno rigida che gli permette di implementare i metodi *put* e *get*. Questa struttura si chiama overlay network ed è una nuova rete "virtuale" che utilizza la rete IP creando una topologia diversa. Alcune DHT, per migliorare l'efficienza del routing, provano a mantenere l'overlay network il più simile possibile alla rete sottostante IP in modo da minimizzare la latenza media dei messaggi scambiati tra i nodi della rete.

1.2 Pastry

Pastry[31] è un protocollo progettato per creare e mantenere un overlay network tale da permette il salvataggio di informazioni e il routing su sistema distribuito. Questo protocollo è usato come livello sottostante a sistemi di storage come PAST[11] o sistemi publish/subscribe come SCRIBE[32]. Ogni nodo di una rete Pastry è caratterizzato da un ID a 128 bit. L'ID indica la posizione del nodo all'interno di uno spazio circolare con 2^{128} indici. Durante il routing, ogni richiesta di un dato con chiave K va inoltrata al vicino il cui ID è numericamente più vicino alla chiave K. La struttura delle tabelle di routing è tale da garantire, in media, un numero di messaggi $O(log_{2^b}N)$ dove b è un parametro che di solito è uguale a 4. Durante il routing gli ID dei nodi e le chiavi sono visti come sequenze di caratteri in base 2^b. Un nodo A inoltra la richiesta di una chiave al nodo il cui ID condivide un prefisso con la

chiave cercata più lungo di quello del nodo A. Se nella tabella di routing non è presente un nodo con queste caratteristiche, la richiesta va inoltrata ad un nodo il cui ID condivide lo stesso prefisso del nodo A ma numericamente più vicino alla chiave.

1.3 La struttura di un nodo

Ogni nodo della rete mantiene in memoria tre tabelle:

- Tabella di routing (**R**)
- Insieme dei nodi foglia (**L**)
- Insieme dei vicini (**N**)

La tabella di routing è composta da $log_{2^b}N$ righe e 2^b colonne. Nella cella della i-esima riga e j-esima colonna è memorizzato un link ad un nodo il cui ID condivide i - 1 caratteri con l'ID del nodo corrente e presenta, come j-esimo carattere, il j-esimo carattere tra quelli disponibili. Per capire meglio analizziamo un esempio dove la tabella di routing di un nodo in un protocollo Pastry che utilizza come identificatori stringhe di 8 bit. Di conseguenza gli ID dei nodi sono scelti nell'intervallo tra 0 e $2^8 - 1$. Come parametro b è stato scelto il valore 2 dunque l'ID di ogni nodo è formato da 4 caratteri in base 4. Mostriamo ora una possibile routing table di un nodo di ID 3120.

0	1	2	3
0213	**1**301	**2**212	**3120**
3 **0**01	**3120**	3 **2**20	3 **3**01
31 **0**1	null	**3120**	31 **3**2
3120	null	312 **2**	null

Come si nota dall'esempio, nella prima riga, cioè quella di indice 0, ci sono tutti i link a nodi il cui ID ha un prefisso comune con il nodo in questione lungo zero caratteri. Nella seconda riga, abbiamo un solo carattere comune e così via fino alla quarta riga nella quale gli ID dei nodi differiscono da quello corrente

solo per l'ultimo carattere. Un'altra cosa che si nota dall'esempio è che non tutte le celle della tabella sono piene. Ciò può significare che è avvenuto un fallimento di un nodo che era in precedenza salvato in una certa posizione, oppure, più semplicemente, che il nodo con un certo ID specifico non esiste. Questo fenomeno è molto probabile che accada nelle ultime righe della tabella dove gli ID devono appartenere a un intervallo molto più ristretto.

L'insieme dei nodi foglia è composto da link ai nodi il cui ID è numericamente più vicino all'ID del nodo corrente. Se l'insieme contiene $|L|$ link, la metà sono collegamenti a nodi con ID minore di quello del nodo selezionato mentre l'altra metà ha un ID maggiore. Vediamo subito un esempio dell'insieme dei nodi foglia del nodo mostrato in precedenza.

LeafSet							
Smaller				Larger			
3032	3101	3102	3103	3121	3130	3132	3200

In questo esempio si memorizzano i collegamenti a 2×2^b nodi anche se sono più frequenti LeafSet con 2^b nodi.

L'insieme dei vicini contiene un elenco dei $|N|$ nodi geograficamente più vicini. La presenza di questo insieme è fondamentale quando vogliamo mantenere un riferimento alla località dei nodi.

1.4 Il routing

Come in tutte le DHT, anche in Pastry una delle funzioni più importanti è il routing. Ogni nodo, per sapere a chi inviare un messaggio di ricerca di una chiave, esegue un algoritmo estremamente semplice nel quale possiamo individuare tre casi diversi.

Il primo caso si ha quando l'ID della chiave da cercare è compreso nell'intervallo dei nodi foglia. Ciò significa che possiamo consegnare il messaggio direttamente al destinatario finale. Ipotizziamo che il nodo presentato nel precedente esempio debba inoltrare un messaggio di richiesta della chiave 3311.

Cercando sull'insieme dei nodi foglia troviamo che il nodo più vicino alla chiave cercata è 3310 e gli inoltriamo il messaggio.

Il secondo caso, quello più probabile, si ha quando la chiave non è presente tra gli ID dei nodi foglia e, sulla tabella di routing si trova un nodo con l'ID che ha un prefisso comune con la chiave più lungo di quello tra la chiave e il nodo locale. Tornando all'esempio visto, ipotizziamo di cercare la chiave 3031. La lunghezza del prefisso comune tra l'ID del nodo (3 120) e la chiave (3 **0**31) è 1 e il primo carattere della chiave diverso dall'ID del nodo è 0. Il nodo cerca sulla tabella di routing il nodo nella posizione [1,0] e invia il messaggio al nodo 3001.

Il terzo caso si verifica quando entrambe le condizioni dei primi due casi sono false. Si cerca tra tutti i nodi che si conoscono quello con l'ID numericamente più vicino alla chiave. Supponiamo ora che il nostro nodo debba cercare la chiave 31 **1**1. Seguendo il ragionamento fatto nel caso precedente, cerchiamo il valore memorizzato in $\mathbf{R}[2,1]$. Purtroppo ci accorgiamo che nella tabella non è memorizzato nessun valore nella posizione cercata. Cerchiamo in ($\mathbf{R} \cup \mathbf{L} \cup \mathbf{N}$) il nodo al quale inoltrare il messaggio e troviamo 3103. In questo caso, poiché la tabella dei vicini non è stata riportata in precedenza, la ricerca è stata fatta solo tra i nodi di $\mathbf{R} \cup \mathbf{L}$. Si noti che, per costruzione dell'insieme dei nodi foglia, si conoscono almeno $|L|/2$ possibili candidati. Ciò ci garantisce la consegna del messaggio a meno di un fallimento simultaneo dei $|L|/2$ nodi della tabella delle foglie con ID minore o maggiore della chiave. Detto questo, possiamo subito notare che, ad ogni passo, la differenza tra la chiave K e l'ID dei nodi ai quali la richiesta va inoltrata decresce. Questa proprietà ci garantisce la terminazione dell'algoritmo di routing. Un'altra cosa da notare è che le ultime righe della tabella di routing sono difficilmente usate. Questo perché, poiché queste righe contengono ID tali che il prefisso in comune con il nodo locale è molto lungo, è molto probabile che gli ID cercati rientrino nell'intervallo dei vicini e quindi il messaggio va inoltrato al nodo con l'ID più vicino alla chiave senza accedere alla tabella di routing.

Per analizzare il costo del routing in termini di messaggi scambiati suppo-

niamo, in prima analisi, che la tabella di routing sia completamente riempita in ogni nodo della rete. Ad ogni passo la lunghezza del prefisso comune tra la chiave e il nodo responsabile della ricerca aumenta di almeno un carattere, cioè 2^b bit. Quindi il numero di messaggi scambiati è proporzionale al numero di caratteri degli identificativi dei nodi, ovvero $O(log_{2^b} N)$. Analizziamo ora il caso in cui l'ID del prossimo hop non è presente in **R**. In questo caso si deve pagare un hop in più per ogni nodo in cui siamo sfortunati. La cosa positiva è che, la probabilità dell'essere "sfortunati" su ogni nodo, con le dimensioni di **L** più usate (2^b e 2×2^b), varia tra 0,02 e 0,006. Nonostante che al caso pessimo il routing costa O(N) messaggi, possiamo trascurare quest'ultima analisi affermando che con alta probabilità il costo del routing è di $O(log_{2^b} N)$.

1.5 L'adattamento alle modifiche delle Rete

Come abbiamo detto in precedenza, una DHT deve essere in grado di auto-organizzarsi per essere tollerante alle modifiche della topologia della rete dovute al fallimento di nodi esistenti oppure all'entrata di nuovi nodi. Analizziamo in questo paragrafo come si comporta Pastry in entrambi i casi. Un nodo, prima di entrare nella rete, deve calcolarsi il proprio ID. Quest'operazione si fa tipicamente calcolando la funzione hash SHA-1 sull'indirizzo IP del nodo. A questo punto deve trovare l'indirizzo di un nodo facente parte della rete che sia geograficamente vicino. Per questo scopo, esistono online degli elenchi di alcuni peer connessi alla rete. Il nodo che deve accedere nella rete, calcola come ID un certo valore X e trova un nodo A già connesso alla rete Pastry. Il nodo A, ricevuta la richiesta di join, avvia la procedura di routing inviando il messaggio di "join" e cercando la chiave X. Il routing termina in un nodo Z con l'ID numericamente più vicino a X. Ogni nodo che riceve il messaggio di "join", invia al nodo X le sue tabelle dello stato. Grazie alle tabelle ricevute, Il nodo X riesce a costruire la propria tabella di routing copiando l'i-esima riga della tabella di routing dall'i-esimo nodo che ha ricevuto la richiesta di "join". Siccome Z è il nodo più vicino a X da un punto di vista degli ID e si assume che

A sia quello geograficamente più vicino, il nodo X copia la tabella dei vicini del nodo A e la tabella delle foglie dal nodo Z. Vediamo ora un esempio di entrata nella rete del nodo 1302. Il routing invia il messaggio di join ai seguenti nodi:

Node 2102	L	L_{A0}	L_{A1}	L_{A2}	L_{A3}
	N	N_{A0}	N_{A1}	N_{A2}	N_{A3}
	R	...	...	...	...
		0231	**1**203	**2**102	**3**123
		...	...	...	...

Node 1203	L	L_{10}	L_{11}	L_{12}	L_{13}
	N	N_{10}	N_{11}	N_{12}	N_{13}
	R	...	...	...	...
		1 **0**31	1 **1**30	1203	1 **3**12
		...	...	...	...

Node 1312	L	L_{20}	L_{21}	L_{22}	L_{23}
	N	N_{20}	N_{21}	N_{22}	N_{23}
	R	...	...	...	...
		13 **0**1	1312	13 **2**1	13 **3**2
		...	...	...	...

Node 1301	L	L_{Z0}	L_{Z1}	L_{Z2}	L_{Z3}
	N	N_{Z0}	N_{Z1}	N_{Z2}	N_{Z3}
	R	...	...	...	...
		null	1301	null	130 **3**
		...	...	...	...

Il nodo 1302 avrà le seguenti tabelle. Come si può vedere dall'esempio, l'insieme dei nodi foglia è stato copiato dal nodo più vicino all'ID scelto dal nodo entrante cioè 1312, l'insieme dei vicini invece è stato copiato dal primo nodo contattato che si assume essere quello geograficamente più vicino. Infine, la tabella di routing contiene una riga della tabella di routing di ogni nodo che riceve un messaggio di "join".

	L	L_{Z0}	L_{Z1}	L_{Z2}	L_{Z3}
	N	N_{A0}	N_{A1}	N_{A2}	N_{A3}
Node 1302	R	**0**231	1302	**2**102	**3**123
		1 **0**31	1 **1**30	1 **2**03	1302
		1302	13 **1**3	13 **2**1	13 **3**2
		null	130 **1**	1302	130 **3**

Alla fine di questa fase, il nuovo nodo invia le proprie tabelle a tutti i nodi presenti nella tabella di routing, nell'insieme dei vicini e quello dei nodi foglia. Finalmente il nuovo nodo può ricevere i messaggi da altri nodi e quindi può partecipare attivamente nella rete Pastry.

L'uscita o il fallimento di un nodo non comporta procedura abbastanza lunga e complessa come nel caso dello join. Infatti, il nodo che deve uscire dalla rete si sconnette senza preoccuparsi di inviare niente ai vicini. I nodi che avevano un riferimento al nodo sconnesso, durante gli invii ping periodici, si accorgono che il nodo non è più presente nella rete e prendono provvedimenti per rimpiazzare il nodo mancante. Se il nodo era linkato da una entrata della tabella delle foglie, si chiede una copia della tabella delle foglie al nodo più estremo dalla stessa parte di quello mancante (minore o maggiore dell'id del nodo). La tabella ricevuta serve a ottenere nuovi link che possono rimpiazzare quelli non più validi. Una cosa simile avviene per la tabella dei vicini mentre, per la tabella di routing, si contatta un nodo nella stessa riga di quello mancante. Se nessun nodo contiene un riferimento ad un nodo che possa rimpiazzare quello mancante, si prova lo stesso procedimento chiedendo ai nodi linkati nella riga successiva.

Capitolo 2

Le vulnerabilità dei sistemi P2P

Le reti P2P sono estremamente vulnerabili ad un ampio spettro di attacchi perché le specifiche dei protocolli sono ottimizzate per garantire la massima efficienza di banda e risorse computazionali ma non tengono in considerazione i problemi relativi alle vulnerabilità. Solo negli ultimi dieci anni si è cominciato a prendere in considerazione i potenziali attacchi che un malintenzionato può effettuare[37]. Inoltre, la natura completamente distribuita di questi sistemi comporta una maggiore difficoltà dell'individuazione e isolamento di potenziali attacchi. Gli avversari che si vogliono affrontare sono utenti del sistema che non seguono correttamente il protocollo al fine di provocare danni agli utenti della DHT, di interrompere o degradare il servizio erogato o di sfruttare le potenzialità della rete per mettere a segno attacchi verso ambienti esterni al sistema. Si assume inoltre che un avversario sia in grado di intercettare e modificare le comunicazioni tra due nodi qualsiasi. Si considera anche la possibilità che i nodi di un gruppo si possano coalizzare per preparare un attacco coordinato e che un singolo utente possa avviare un elevato numero di nodi in una sola macchina.

Negli ultimi anni in letteratura ci si è molto occupati di studiare le caratteristiche degli attacchi volti a destabilizzare il servizio offerto da una rete P2P. Essendo la materia molto vasta ed il numero di pubblicazioni molto elevato, non esiste ancora una totale convergenza su un'unica classificazione degli

attacchi. Sit e Morris[34] analizzarono per primi il problema della sicurezza di una generica DHT fornendo una prima classificazione dei potenziali attacchi e delle linee guida per limitarli. Gli articoli pubblicati successivamente si sono praticamente basati sulla classificazione di Sit e Morris fornendo nuove potenziali soluzioni sempre più raffinate.

In questa tesi classificheremo gli attacchi nelle seguenti classi:

Attacchi Sybil dove un attaccante introduce un elevato numero di nodi maliziosi al fine di controllare parte del traffico della rete;

Attacchi allo storage dove uno o più nodi maligni non eseguono correttamente il protocollo con l'obiettivo di negare l'accesso alle risorse richieste da altri peer;

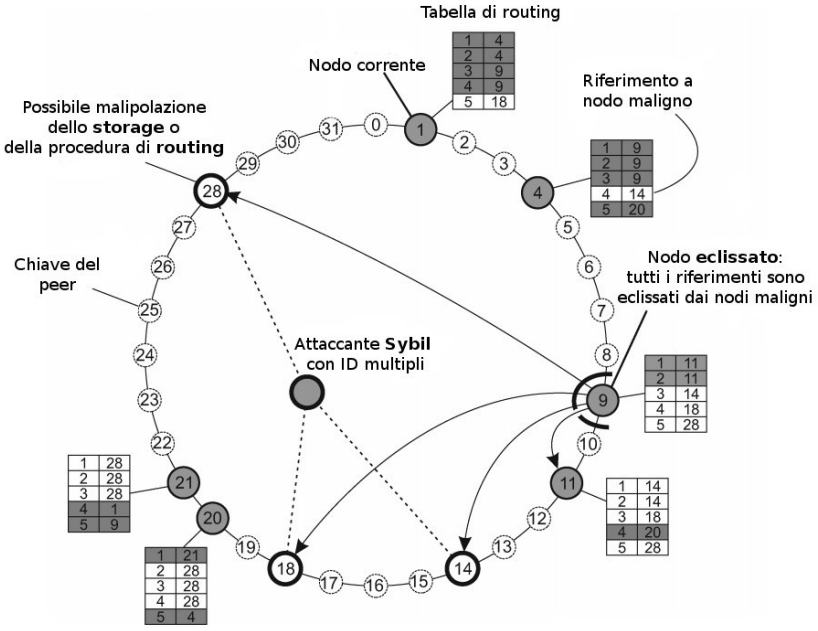
Attacchi al routing dove un malintenzionato inietta delle informazioni fuorvianti al fine di negare l'accesso ad alcune risorse o nascondendo parti della rete (attacco Eclipse);

Attachi di Denial of Service nei quali un attaccante agisce in modo da compromettere il funzionamento della DHT (DoS) oppure sfrutta l'elevato numero di nodi di una rete P2P per generare una mole di messaggi tutti diretti verso un unico target al fine di obbligarlo a servire richieste fittizie invece svolgere il proprio dovere (DDoS);

Attacchi Man in The Middle attraverso il quale un attaccante è in grado di intercettare e modificare il contenuto dei messaggi scambiati da una generica coppia di nodi.

Come si può osservare in figura 2.1, i suddetti attacchi vengono spesso combinati tra loro per aumentare gli effetti dannosi. Nell'esempio viene mostrato che, attraverso un attacco Sybil, si facilita la messa a punto di un attacco Eclipse grazie al quale è possibile intercettare tutti i messaggi in uscita dal nodo eclissato rendendo possibile attacchi di negazione del servizio di routing e storage.

Figura 2.1: Combinazione degli attacchi Sybil, Eclipse e negazione del servizio di Storage e Routing

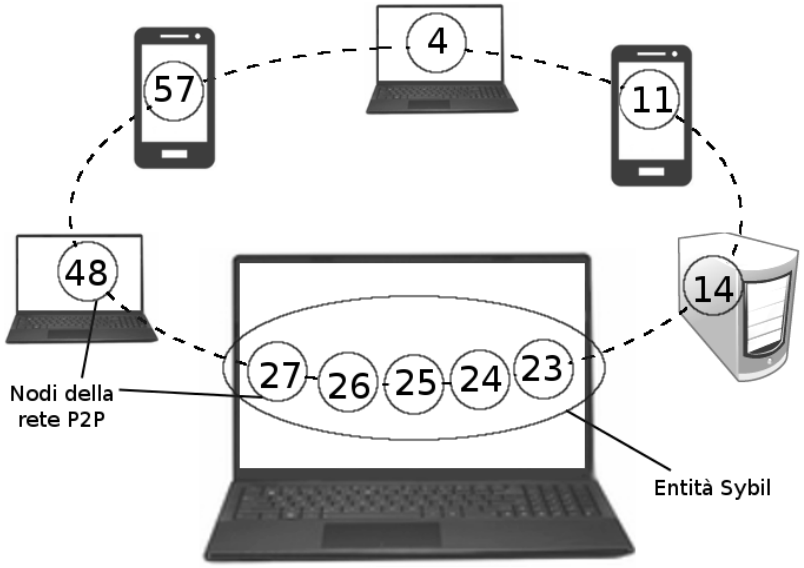


L'obiettivo di questo capitolo è quello di presentare gli attacchi più comuni che possono minacciare una generica rete P2P analizzando gli effetti e presentando delle potenziali contromisure necessarie per far fronte ai suddetti attacchi.

2.1 Attacchi Sybil

Il nome "Sybil" è tratto dal libro di Flora Rheta Screiber in cui si parla di una donna con molteplici personalità. Un attacco Sybil, infatti, consiste nella creazione di un insieme più o meno numeroso di entità controllate da un'unico soggetto. Queste entità, che successivamente verranno indicate come "entità Sybil", possono risiedere su una sola macchina oppure possono essere distribuite sulla rete geografica.

Figura 2.2: Rete P2P con la presenza di alcune entità Sybil



2.1.1 La vulnerabilità

In un sistema P2P, il valore del $NodeId$ che viene assegnato ad un peer è solitamente determinato dal peer stesso calcolando una funzione hash dell'indirizzo IP e del numero di porta. Nella maggior parte dei casi non esiste un'unità centrale fidata che certifichi l'associazione la tra la macchina (o utente) e il valore $NodeId$. Per questo motivo è possibile per un attaccante generare più nodi su una stessa macchina assegnando loro dei $NodeId$ con valore numerico simile riuscendo quindi a controllare completamente una parte della rete. La figura 2.2 mostra un esempio di una rete con la presenza di entità Sybil le quali si assegnano tutte le chiavi comprese tra 23 e 27. Stando alle politiche della DHT per l'assegnazione degli oggetti corrispondenti ad una determinata chiave e delle relative repliche, tutti questi dati potrebbero risiedere in una sola macchina trasformandola in un *single point of failure*. Di conseguenza, un simile attacco potrebbe essere dannoso anche se non combinato con altre tecniche. Il problema assume dimensioni sproporzionate se le entità Sybil vengono utilizzate per aumentare l'efficacia di altri attacchi e provocare una destabilizzazione del servizio preso di mira.

2.1.2 Le possibili soluzioni

Per limitare questo tipo di attacchi è necessario assicurarsi che $NodeId$ diversi corrispondano a nodi diversi. Un peer può acquisire informazioni utili per individuare nodi contraffatti utilizzando dati provenienti da altri peer oppure sfruttando informazioni derivanti da una terza parte fidata. Vediamo ora le soluzioni completamente decentralizzate che sono state proposte in letteratura in questi ultimi anni analizzando i motivi per cui non sono mai state applicate su sistemi funzionanti. Queste soluzioni sfruttano delle euristiche che impongono dei limiti sul numero di accessi alla rete o sfruttano dei limiti esistenti come la capacità di calcolo degli elaboratori oppure i rapporti sociali tra gli utenti.

Limite di nodi con lo stesso IP

Una possibile soluzione fu proposta da Dinger e Hartenstein[10] i quali proponevano un sistema distribuito per limitare il numero di nodi che si connettevano dallo stesso indirizzo IP. Tuttavia questa soluzione non può essere applicata perché non limita attacchi Sybil effettuati con un numero elevato di indirizzi IP. Inoltre esistono degli ISP che forniscono ai clienti finali una connessione ad internet all'interno di un router NAT. In questo caso potremmo avere anche centinaia di nodi onesti che si connettono ad internet utilizzando lo stesso indirizzo IP. Ovviamente non possiamo estromettere questa tipologia di utenti e non avrebbe nemmeno senso imporre un limite di migliaia di utenti con lo stesso indirizzo IP.

Limite di nodi geograficamente vicini

Altre soluzioni[1] sfruttano il fatto che le entità Sybil si connettono spesso in una sola locazione geografica oppure in locazioni vicine. In questo caso si pone un limite al numero di peer all'interno di un'area geografica. Questa soluzione è difficile da attuare in quanto, data la natura completamente distribuita, non possiamo essere sicuri che la locazione dichiarata dai peer sia quella reale. Tanti peer istanziati su una stessa macchina po-

trebbero dichiarare diverse locazioni dislocate su tutto il globo byassando così un potenziale limite imposto.

Limite della capacità computazionale

Per limitare il numero di nodi che si connettono dalla stessa macchina è stato proposto di assegnare la risoluzione di un puzzle computazionale ai peer quando si connettono. Questa idea[2] si basa sul fatto che le risorse di calcolo di una macchina sono limitate e quindi è sufficiente imporre un limite di tempo necessario per risolvere un solo puzzle per evitare che su una sola macchina vengano installati più nodi. Questa soluzione sarebbe applicabile solo nel caso in cui la rete Internet fosse composta solo da macchine con la stessa potenza computazionale. Come sappiamo la rete internet è composta da un numero eterogeneo di dispositivi con capacità diverse. Infatti, quello che potrebbe risultare un task computazionalmente impossibile per uno smartphone può essere facilmente calcolabile con un supercomputer.

Limite dei rapporti interpersonali

Come abbiamo potuto constatare queste soluzioni tendono a limitare gli attacchi Sybil ma non riescono ad eliminarli. Al momento le uniche tecniche che riescono ad arginare con efficacia le entità Sybil si basano sull'utilizzo di reti sociali. Queste soluzioni sono anche completamente decentralizzate ed altamente scalabili. L'assunzione[43] che sta alla base di questo approccio sfrutta il fatto che ogni nodo Sybil non può essere associato ad una persona reale perché una persona non riuscire a creare e gestire contemporaneamente un numero di identità tali che i relativi peer possano portare a termine un attacco Sybil.

Queste soluzioni limitano indirettamente il numero di edge che collegano i nodi Sybil ai nodi onesti limitando quindi l'efficacia di un attacco di questo tipo. L'unico difetto di questo approccio consiste nel fatto che, da un punto di vista architetturale, il livello della DHT deve essere strettamente dipendente dall'applicazione che verrà implementata utilizzando

la DHT stessa. Oltre a questo, il range di applicazioni implementabili utilizzando questa tecnica è limitato dai servizi basati su reti "sociali" come la messaggistica.

Riassumendo quanto detto, un peer, avendo una visione locale limitata all'interno di una DHT, non riesce a riconoscere ed eliminare completamente gli attacchi di tipo Sybil.

Analizziamo ora una rete P2P nella quale è presente un'entità centralizzata e fidata che certifica l'associazione di un nodo con una sola macchina e una sola persona fisica. Una soluzione proposta in letteratura è quella di creare un sistema di firme digitali con certificati garantiti dall'unità centralizzata. Tra le differenti politiche che utilizzano un approccio di questo tipo, vi è anche l'idea di far pagare l'utente per la creazione del certificato. In questo modo, anche se un attaccante riuscisse a bypassare tutti i sistemi di sicurezza attuati in fase di registrazione, riuscirebbe a creare un numero elevato di nodi a fronte di una spesa non indifferente. Inoltre si potrebbero sfruttare i movimenti di denaro provenienti da un gruppo ristretto di conti correnti o carte di credito per poter localizzare l'insieme di nodi potenzialmente pericolosi.

2.2 Attacchi contro lo storage

Le reti esistenti non offrono nessuno strumento di verifica per controllare che i contenuti memorizzati non siano fasulli o addirittura dannosi. Esistono quindi attacchi che tendono ad inserire nel sistema P2P riferimenti a file o nodi insistenti oppure corrotti. Possiamo classificare queste azioni in tre principali categorie.

2.2.1 Content pollution

In una DHT il contenuto identificato da una determinata chiave k è assegnato ad alcuni nodi che hanno il compito di mantenere la ridondanza del contenuto e di fornirlo ai nodi che ne facciano richiesta. L'avvelenamento dei contenuti è

usato in reti che utilizzano un doppio livello di indirizzamento. L'attacco consiste nell'inserimento massiccio di oggetti che, in corrispondenza della chiave k, contengono riferimenti ad altre chiavi inesistenti o nodi che non possiedono il file cercato.

É applicato maggiormente nei sistemi di file sharing perché consiste nel diffondere riferimenti a file inesistenti o che non corrispondono ai metadati pubblicati. In passato, dopo l'avvento di reti di file sharing completamente distribuite, la diffusione di contenuti multimediali protetti da copyright non poteva essere più contrastata efficacemente passando per vie legali. Per questo motivo l'industria musicale attuò questo tipo di attacco per scoraggiare l'utilizzo di sistemi come KaZaA.

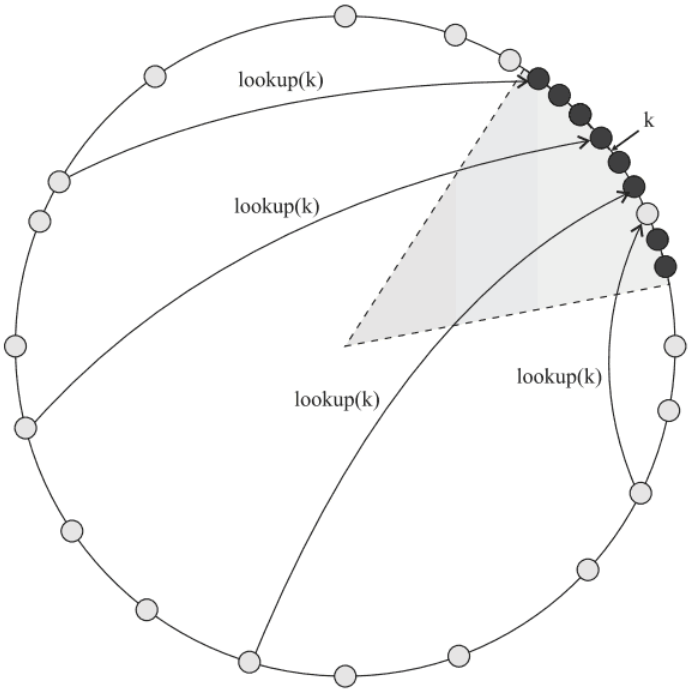
Un attacco di questo tipo, nonostante la sua efficacia, non richiede particolari capacità computazionali o di banda. Per limitare questa azione è necessario che tutti i dati emessi siano firmati in modo da poter sempre risalire al responsabile di un determinato inserimento. Il problema si risolve implementando un sistema che esclude dalla rete i nodi che hanno inserito dati fittizi.

2.2.2 Node insertion attack

L'inserimento di nodi è un attacco volto ad oscurare i contenuti memorizzati nella DHT e consiste nel generare un numero di nodi con $NodeId$ vicino ad una chiave k che si vuole rendere non reperibile come mostrato in figura 2.3. Durante la procedura di routing, un potenziale richiedente, con alta probabilità inoltrerà la richiesta ad uno di questi nodi generati ad-hoc dall'attaccante il quale, dichiarerà di non possedere il contenuto ricercato.

Una prima precauzione da prendere contro gli attacchi di questo tipo è quella di evitare che un nodo possa generare un $NodeId$ in modo da evitare la situazione in cui i nodi maliziosi popolino una zona specifica del keyspace. Infatti risulta necessario avere un'unità fidata che assegna un'identità casuale e certificata ad ogni peer. Un'altra difesa proposta in letteratura consiste nell'utilizzo di tabelle di routing che impongono forti vincoli sull'insieme dei vicini. Ogni entry della routing table deve contenere il riferimento al nodo

Figura 2.3: Rete P2P con la presenza di nodi inseriti in modo da oscurare la chiave k



conosciuto più vicino ad un determinato punto del keyspace. Un esempio di tabella di routing vincolata è la *finger table* di chord la quale, nell'$i-esima$ posizione contiene il riferimento al nodo con l'ID più vicino al punto $n + 2^{i-1}$ dove n è l'ID del nodo che ospita la tabella stessa. Non potendo generare arbitrariamente $NodeId$ tali da rispettare tutti i vincoli imposti, risulta arduo per un attaccante poter inquinare la tabella di routing di un nodo vittima. Un approccio diverso utilizzato per limitare l'inquinamento della routing table consiste nel refresh periodico della stessa. Alcune implementazioni combinano il refresh della tabella di routing con un limite del tasso di aggiornamento, in modo che un attaccante non riesca ad eseguire un iniezione massiccia perché il tasso di aggiornamento è limitato. Inoltre dopo un certo lasso di tempo la tabella del nodo vittima si aggiorna eliminando tutti i riferimenti agli altri nodi tra cui quelli ai nodi corrotti.

2.2.3 Possibili stategie di storage ridondante

Una volta progettata una rete P2P dove un'unità centrale assegna ai nodi un ID casuale e una coppia di chiavi pubblica e privata per firmare i documenti, ci dobbiamo preoccupare solo della presenza di nodi corrotti che non forniscano le informazioni richieste. In letteratura il problema si risolve semplicemente adottando tecniche di routing e storage ridondante. Per risolvere il problema dello storage esistono diverse possibilità per memorizzare le repliche di un contenuto.

Memorizzare nei nodi geograficamente più vicini

Data la vicinanza dei peer, c'è una maggiore semplicità nel mantenere efficacemente un determinato grado di ridondanza di un oggetto. Il problema sorge se, per un qualsiasi problema, anche derivante da cause naturali, una certa area geografica rimane isolata dal resto della rete. In questi casi tutte le chiavi memorizzate dai nodi della zona andranno perse.

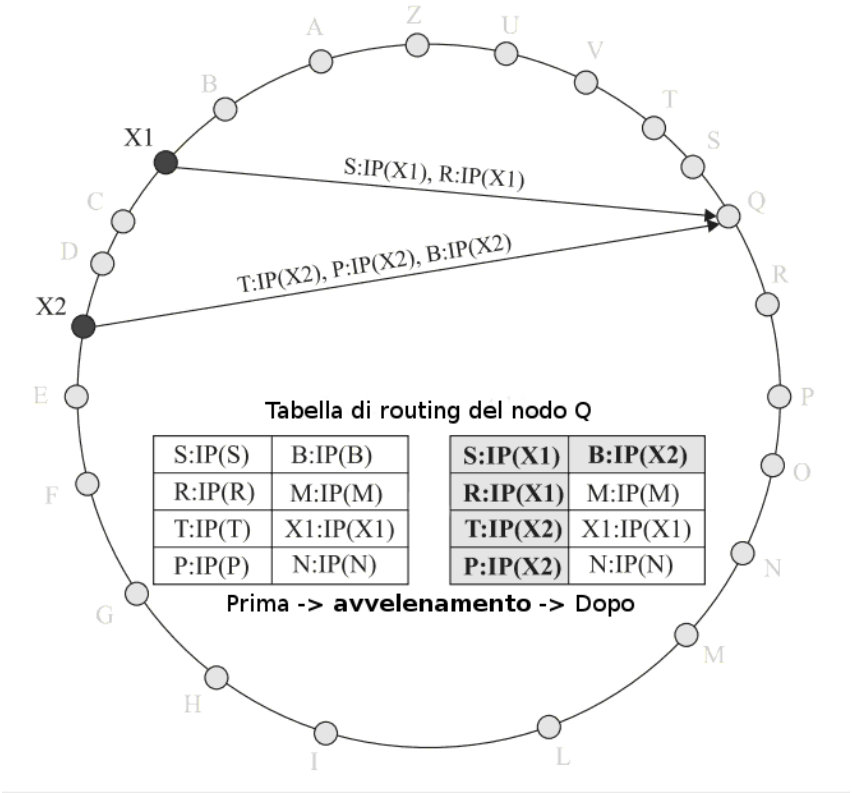
Memorizzare nei nodi più vicini nel keyspace

Memorizzare le repliche di un contenuto in nodi vicini del del keyspace complica il mantenimento di copie consistenti senza apportare un effettivo miglioramento alla resistenza di fronte ad un potenziale attaccante. Visto che un protocollo di questo tipo è pubblico, attraverso un attacco Sybil, nodi malevoli possono riuscire ad oscurare gli indirizzi del keyspace dove risiedono tutte le repliche di un oggetto.

2.3 Attacchi al routing

Un potenziale attaccante potrebbe compromettere il corretto funzionamento della procedura di routing iniettando dei riferimenti generati ad-hoc nelle routing table dei nodi vittima. In questo modo, il comportamento di un nodo vittima colpisce involontariamente le procedure di storage dei contenuti e forwarding dei messaggi.

Figura 2.4: Esempio di avvelenamento della routing table

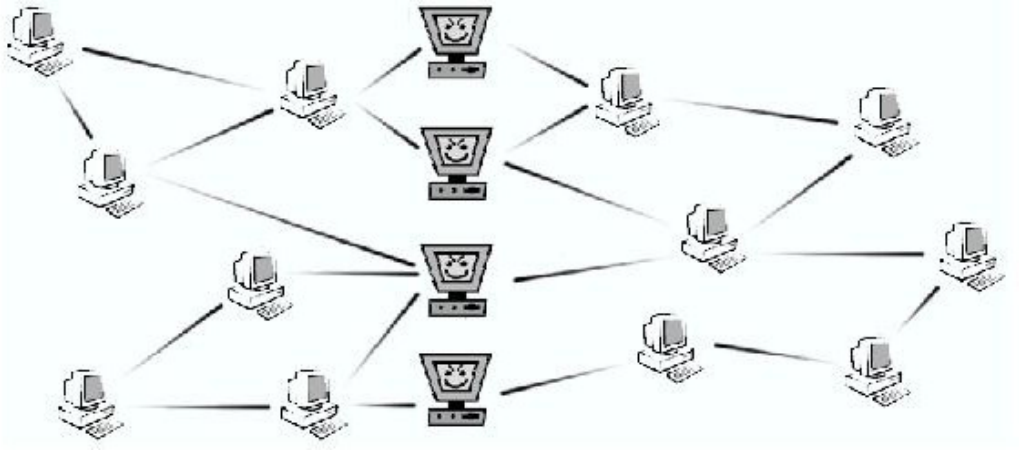


2.3.1 Avvelenamento della tabella di routing

Molte DHT utilizzano un approccio *push-based* per rinnovare le tabelle di routing. Ciò significa che i messaggi provenienti da altri nodi, come pubblicazioni delle routing table o messaggi di lookup provenienti da terze parti, forniscono dati per l'aggiornamento dello stato di un nodo. Sapendo questo, per un malintenzionato diventa molto facile iniettare informazioni nella tabella di routing di un nodo vittima al fine di dirigere i propri messaggi verso un nodo "corrotto". La figura 2.4 mostra come due nodi $X1$ e $X2$ riescano ad iniettare in un terzo nodo Q dei riferimenti che puntano a loro stessi compromettendo anche il corretto funzionamento di questo ultimo nodo. É evidente che la presenza di un elevato numero di nodi Sybil può facilitare la buona riuscita di un attacco di questo tipo.

In linea teorica una DHT può utilizzare un approcio *pull-based* in cui l'ag-

Figura 2.5: Eclipse Attack



giornamento della routing table avviene attraverso una procedura di polling del nodo stesso nei confronti di altre entità. Seguendo questo secondo approccio, le DHT dovrebbero essere più resistenti ad avvelenamenti della routing table. Questa tecnica però comporta un alto numero di messaggi infatti, ad oggi, non si conoscono DHT che abbiano adottato esclusivamente questo schema.

2.3.2 Attacchi Eclipse

Un attacco di tipo Eclipse, anche conoscuto in letteratura come *partition attack*, è una forma particolare dell'avvelenamento della routing table. Questo mira a separare un insieme di nodi che partecipano al protocollo P2P dal resto dell'overlay network come si può vedere in figura 2.5. L'attaccante raggiunge il proprio scopo quando la routing table di un nodo vittima risulterà contenere solo riferimenti a nodi malevoli. Se questa tecnica va in porto con successo, un attaccante può controllare tutto il traffico in uscita da un nodo vittima "eclissandolo" così dal resto della rete.

Anche questi attacchi alla procedura di routing si limitano proibendo ai nodi di generare il proprio $NodeId$ attribuendo questo compito ad un'unità centralizzata. I messaggi scambiati per rinnovare la routing table devono essere firmati in modo da poter sempre risalire al responsabile di un determinato inserimento di un nodo.

2.3.3 Il routing ridondante

Combinando molti degli attacchi visti in precedenza, è molto probabile che un potenziale attaccante provi a rendere irreperibili i contenuti della DHT. La tecnica più usata per provare a combattere questo fenomeno consiste nel routing ridondante[38].

Analizziamo in un primo momento i tre possibili modi per implementare il routing in un sistema distribuito. In seguito indichiamo con *I (Initiator)* il nodo che esegue la procedura di routing per ricercare o memorizzare una chiave.

Routing iterativo

Nel routing iterativo (figura 2.6) il nodo I nodo esegue direttamente tutta la procedura di ricerca della chiave. Ad ogni passo, I, basandosi sulle regole del protocollo, contatta un determinato nodo il quale fornisce ad I il riferimento ad il nodo successivo nella procedura di routing. Questa soluzione impiega un elevato numero di messaggi ($2 \times NumHops$) in modo sequenziale però, l'*Initiator* è sempre al corrente dello stato della procedura e, se questa si interrompe, conosce qual'è il nodo responsabile dell'interruzione.

Routing ricorsivo

Nel routing ricorsivo (figura 2.7) il nodo I si aspetta di ricevere come risposta direttamente il valore cercato. Ogni nodo intermedio si incarica di portare avanti la procedura cercando sulla tabella di routing il prossimo nodo da contattare e inoltrando la richiesta al suddetto nodo. In questa soluzione il numero di messaggi dipende da come la risposta viene inviata all'*Initiator*. Se la risposta viene inviata direttamente ad I, il numero totale dei messaggi sarà $NumHops + 1$ altrimenti, se la risposta segue il percorso inverso della query, il numero di messaggi diventerà $2 \times NumHops$. Nel primo caso nessun nodo riceve un feedback sull'esito della procedura quindi, in caso di presenza di un nodo corrotto, nessun nodo potrebbe individuare un potenziale "indiziato". Nell'altro caso, grazie

ai feedback che vengono ricevuti come risposta della procedura ricorsiva, i nodi vengono a conoscenza di quale peer ha interrotto la procedura.

Routing tracciato

Il routing tracciato (figura 2.8) consiste in una soluzione intermedia tre le due viste in precedenza. Ogni nodo intermedio inoltra la richiesta al prossimo nodo e, in parallelo, invia ad I il riferimento al nodo contattato. Nonostante il numero di messaggi inviati in questa soluzione rimanga comunque elevato ($2 \times NumHops$), il tempo per eseguire la query si riduce perché il messaggio di forward e quello di feedback vengono inviati in parallelo. Questa soluzione offre un buon compromesso tra velocità di esecuzione e controllo da parte del nodo *Initiator*.

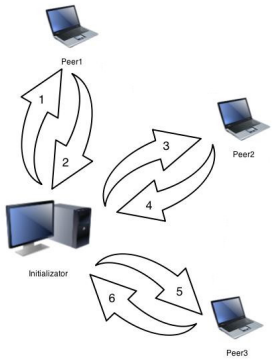


Figura 2.6: Routing Iterativo

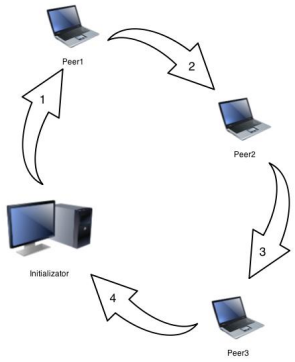


Figura 2.7: Routing Ricorsivo

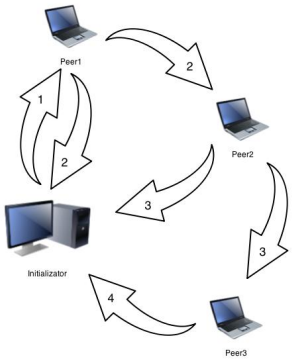


Figura 2.8: Routing Tracciato

Sapere quale nodo interrompe la procedura di routing è importante per avere un sospettato in caso si pensi che il processo sia fallito a causa dell'intromissione di qualche malintenzionato. Se si è sicuri che una chiave esista sulla DHT e il routing non la trova ci possiamo trovare di fronte a una delle sequenti possibilità:

- Per un errore accidentale la tabella di routing del nodo che ha interrotto la procedura è stata danneggiata;

- Il nodo che ha interrotto la procedura è stata vittima di un attacco di inquinamento della tabella di routing;
- Il nodo che ha interrotto la procedura è un attaccante e l'ha fatto consapevolmente per destabilizzare il sistema.

Una delle possibilità per individuare una delle situazione sopra elencate consiste nell'utilizzo del routing ridondante. Se un nodo inoltra una query lungo tre percorsi, se da due percorsi arriva il risultato atteso mentre dal terzo la query fallisce, è facile intuire che su questo ultimo percorso c'è stato un problema. In letteratura possiamo trovare due diverse tipologie di routing ridondante:

Il routing multiple-path , presentato per la prima volta in [3], consiste nell'inviare una richiesta su percorsi distinti diretti alle diverse repliche del contenuto richiesto.

Il routing wide-path, presentato da Hildrum and Kubiatowicz[22], è una forma di routing iterativo dove ad ogni passo, il nodo interpellato risponde con una lista degli r nodi, tra quelli conoscuiti, più vicini alla chiave k.

Mentre il multiple-path è più appropriato se le repliche del contenuto sono sparse per il keyspace, il wide-path è migliore se le repliche sono salvate nei peer numericamente più vicini. Inoltre, studi statistici e simulazioni mostrano che, in presenza di un'alta percentuale di nodi maligni, il wide-path è molto più affidabile del multiple-path.

2.4 Altri attacchi

Per un peer malintezionato può risultare particolarmente semplice sfruttare le potenzialità della DHT per negare il servizio offerto da un sistema interno o esterno alla rete P2P. Il fatto che un nodo non si comporti in modo "collaborativo" negando la conoscenza di una determinata chiave richiesta, può essere classificato come un attacco di *Denial of Service (DoS)*. Ovviamente in questo esempio si va a destabilizzare il servizio offerto dall'applicazione che sfrutta

la DHT attaccata. Di seguito presenteremo altri esempi di attacchi di DoS effettuati contro la DHT stessa (Join e leave rapidi) oppure un sistema esterno (DDos)

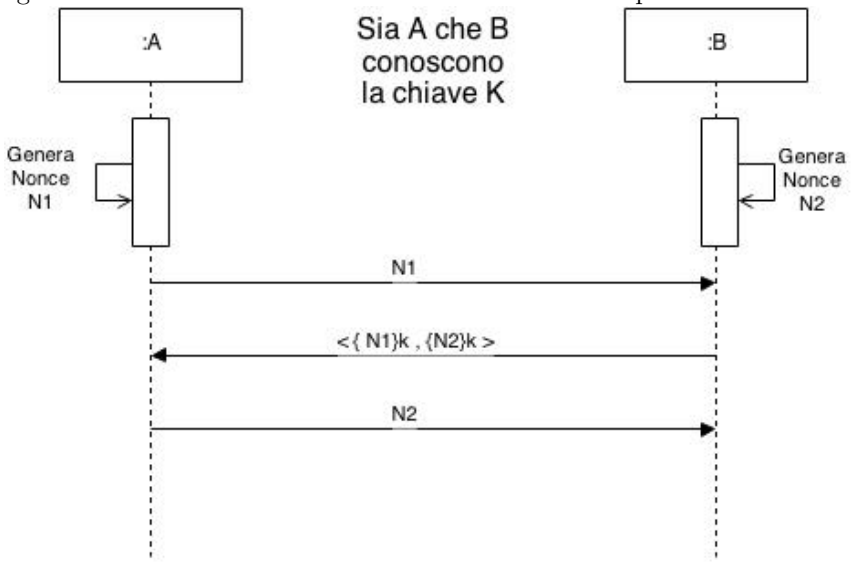
2.4.1 Join e leave rapidi

Al momento in cui un nodo entra a far parte di una rete P2P oppure la abbandona, secondo il protocollo previsto dalla DHT si procede con uno scambio di informazioni necessarie per mantenere un certo grado di ridondanza dei contenuti. Il fallimento di un elevato numero di nodi provoca un notevole scambio di messaggi e la presenza di tabelle di routing inconsistenti con il conseguente degrado delle prestazioni. Eseguendo ripetutamente questa procedura, il sistema sarà sottoposto ad un elevato *churn rate* mettendo in atto un continuo scambio di informazioni che potrebbero congestionare la rete. Ovviamente per creare un attacco così efficace su una rete di scala globale, è necessario che il numero di artefici sia veramente elevato e ciò è possibile sfruttando la presenza di numerosi nodi Sybil. Di conseguenza, per limitare un attacco DoS di questo tipo si possono applicare le stesse tecniche viste in precedenza per arginare gli attacchi Sybil.

2.4.2 Distributed Denial of Service (DDoS)

Un nodo maligno, utilizzando semplicemente una variante dell'avvelenamento dell'indice, può riuscire a indirizzare un'enorme mole di messaggi verso un'entità vittima interna o esterna alla rete. É possibile infatti inserire una chiave corrispondente ad un contenuto molto popolare indicando come riferimento l'indirizzo IP della potenziale vittima. Se il protocollo distribuito non verifica la corrispondenza tra la chiave di un contenuto e la fonte del contenuto stesso, la conoscenza di questa informazione sarà mantenuta provocando un continuo bersagliamento del nodo vittima. Anche in questo caso, per limitare gli effetti di un attacco di questo tipo, è necessario eliminare la possibilità di attacchi allo storage introducendo meccanismi di firma digitale.

Figura 2.9: Protocollo che utilizza la tecnica dei nonce per l'autenticazione

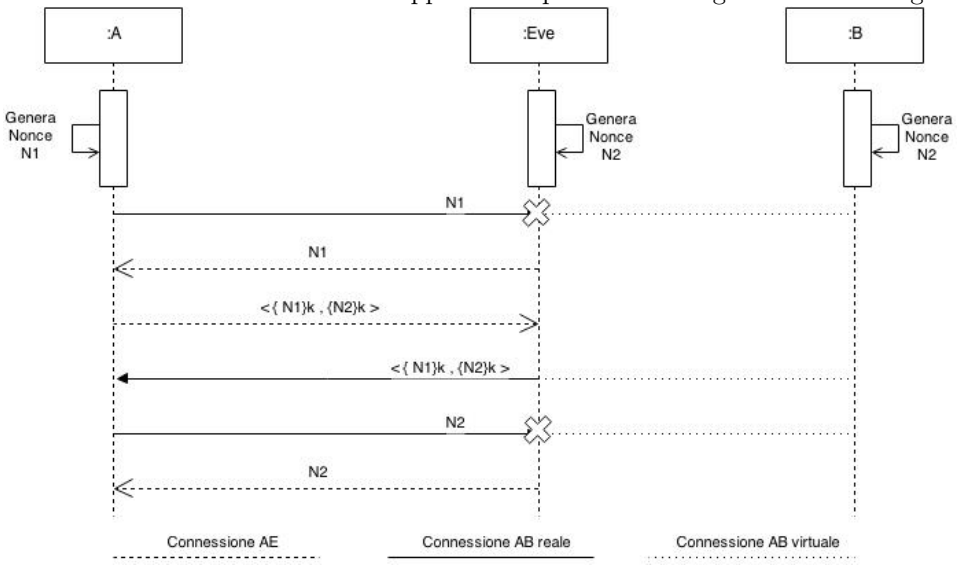


2.5 Attacchi Man In the Middle (MITM)

Un *Man In The Middle* consiste in un attacco nel quale un malintenzionato si intromette nella comunicazione tra una coppia generica di nodi potendo leggere o modificare le informazioni che vengono scambiate. Le parti interessate possono continuare a comunicare inconsapevoli che il collegamento tra loro è compromesso da una terza parte. In ottica della sicurezza sul P2P, gli attachi MITM sono un pericolo presente nei casi reali. Prendiamo in considerazione reti come Kad le quali utilizzano un sistema di *buddy* per avviare una comunicazione verso i nodi dietro al NAT. In Kad ogni nodo sceglie un nodo buddy verso il quale stabilisce una connessione sempre attiva. Le richieste di ricerca di un contenuto vengono inviate al nodo buddy il quale le inoltra al nodo interessato. Un attaccante in grado di controllare un buddy potrebbe mettere a segno un attacco MITM praticamente senza sforzo. Per ovviare a questa situazione, ogni coppia di nodi, prima di avviare una comunicazione, utilizza protocolli di autenticazione e generazione di chiavi crittografiche tra gli endpoint per garantire la riservatezza della comunicazione.

Molti di questi protocolli, però, sono basati sulla trasmissione di *nonce* che

Figura 2.10: Attacco Parallel Session applicato al protocollo raffigurato nell'immagine 2.9



generano altri bug di sicurezza. Un *nonce* non è altro che una stringa casuale generata da un'entità per poter verificare l'identità dell'interlocutore il quale deve cifrare il *nonce* con una certa chiave segreta. Analizziamo per esempio il protocollo della figura 2.9 nel quale due nodi A e B sono a conoscenza di una chiave comune K. A genera un nonce $N1$ e lo invia a B il quale, una volta ricevuto il messaggio genera un nonce $N2$, cifra sia $N1$ che $N2$ utilizzando la chiave K e invia ad A le stringhe cifrate. Anche A conosce la chiave K, quindi può controllare che la correttezza di $N1$ cifrato e può decifrare il nonce $N2$ da trasmettere in chiaro a B. Il protocollo appena presentato è passibile di attacchi MITM classificabili in due categorie:

chosen chipertext attack tramite il quale un attaccante fornisce ad un nodo vittima dei nonce da cifrare generati ad-hoc in modo da ottenere un testo cifrato corrispondente ad un testo noto. Iterando più volte questo procedimento si riesce a dedurre la chiave privata in possesso della vittima.

interleaving attack nel quale si interlacciano più istanze del protocollo di autenticazione con uno o più nodi vittima. L'obiettivo è quello di ottenere

da una sessione le informazioni necessarie per autenticarsi nell'altra e viceversa.

Un esempio di interleaving attack è il *parallel session attack* nel quale un attaccante crea due sessioni parallele verso un solo nodo vittima. Ipotizziamo di voler attaccare il protocollo presentato precedentemente e raffigurato nell'immagine 2.9. La figura 2.10 mostra che un Man in The Middle E senza essere a conoscenza della chiave K, potrebbe autenticarsi con A utilizzando due sessioni parallele. É necessario che E riesca ad intercettare il nonce $N1$ creato da A lungo la connessione AB e crei una nuova sesssione di autenticazione EA inviando ad A proprio $N1$. Quest'ultimo risponderà con la coppia $< \{N1\}_K, \{N2\}_K >$ sulla connessione EA ad E il quale, spacciandosi per B, inoltrerà il messaggio ad A lungo la connessione AB. A, essendo sicuro di comunicare con B, risponde con il nonce $N2$ decifrato. Anche in questo caso E intercetta il messaggio e lo inoltra ad A sulla connessione EA terminando con successo quest'ultima sessione di autenticazione.

Uno studio dell'IBM mostra che, per resistere ad attacchi di tipo Man In The Middle, è necessario che il protocollo di handshake sia ad almeno tre vie. Inoltre, i messaggi cifrati devono dipendere dalla direzione nella quale vengono inviati. Ciò significa che un messaggio cifrato da A deve essere distinguibile da quello cifrato da B.

Capitolo 3

Tecniche di crittografia

Creare delle comunicazioni sicure non è certamente una necessità degli ultimi anni, già nell'antichità si utilizzavano dei cifrari per proteggere dei messaggi segreti. Come si riporta in[15], il metodo più antico di cui si ha notizia fu inventato dagli spartani nel V secolo a.C.. Si utilizzavano due aste cilindriche identiche: una in possesso del mittente e l'altra in possesso del destinatario. La cifratura consisteva nell'avvolgere a elica una fettuccia di cuoio nella quale si scriveva il messaggio. La fettuccia era quindi svolta e spedita al destinatario il quale poteva interpretare il messaggio semplicemente riavvolgendo la fettuccia nella propria asta e ricostruendo così i giusti accostamenti tra le lettere. Uno dei più grandi crittografi del passato è probabilmente Giulio Cesare il quale utilizzava un cifrario che consisteva nella sostituzione di ogni lettera del messaggio in chiaro con quella che segue tre posizioni più avanti nell'alfabeto che si assume circolare.

Passando alla storia più recente, uno dei cifrari più importanti è stato quello utilizzato dai tedeschi durante la seconda guerra mondiale. Si trattava di una macchina da scrivere al cui interno erano situati tre dischi con 26 contatti per lato (uno per ogni lettera dell'alfabeto tedesco). I dischi ruotavano con velocità diverse: il primo ruotava ad ogni lettera inserita, il secondo ruotava al completamento di un giro da parte del primo e il terzo al compimento di un giro da parte del secondo. I contatti sui lati dei dischi permettevano di associare

la lettera premuta sulla tastiera con un'altra determinata dalla posizione dei dischi. La sicurezza del cifrario derivava dal fatto che le associazioni tra lettere cambiavano durante il testo e quindi diventava molto più difficile decifrare il messaggio senza avere la macchina enigma.

Dopo questo breve "excursus storico" andiamo ad analizzare come la crittografia si sia evoluta nei giorni nostri. In un primo momento analizzeremo la cifratura simmetrica ovvero l'estensione diretta dei cifrari storici. In un secondo momento analizzeremo tecniche più moderne e complesse rese necessarie dalla crescente necessità di comunicare in maniera sicura anche con persone molto distanti o mai incontrate con le quali è impossibile accordarsi su chiavi di cifratura. Vedremo infatti i cifrari a chiavi pubblica e il funzionamento della firma digitale. Infine vedremo una tecnica ancora più moderna che, nonostante non sia ancora molto utilizzata, è molto potente. La tecnica serve a dimostrare un'affermazione senza però rivelare nient'altro oltre alla veridicità della stessa. Stiamo parlando delle dimostrazioni a conoscenza zero (zero-knowledge).

3.1 Cifratura simmetrica

I cifrari storici di cui si è riportato un brevissimo sunto, funzionano se mittente e destinatario si trovano nelle medesime condizioni. Vale a dire possedere lo stesso bastone cilindrico, conoscere lo stesso algoritmo oppure possedere la stessa macchina inizializzata con la stessa posizione dei dischi. Per questo motivo, i cifrari storici, come altri cifrari che vedremo in questo paragrafo, sono classificati come simmetrici.

Il padre dei cifrari moderni è One-time-pad. Esso si basa su un or esclusivo bit a bit tra due sequenze binarie derivati dal testo in chiaro e da una chiave casuale. Il risultato è una sequenza binaria dove il bit in posizione i è uguale a 1 se e solo se i due bit in posizione i delle due sequenze confrontate sono diversi. Con questo cifrario è possibile rendere pubblico l'algoritmo e tenere segreta soltanto la chiave.

A condizione che la chiave sia casuale e lunga almeno quanto la stringa da cifrare, si può dimostrare che One-time-pad è un cifrario perfetto. Dimostrare la perfezione di un certificato equivale a dimostrare che la probabilità che il messaggio originale M sia un certo messaggio m non deve cambiare anche conoscendo la probabilità che la stringa cifrata C sia una determinata stringa c. In termini probabilistici questa affermazione si scrive come

$$\mathcal{P}(M = m) = \mathcal{P}(M = m|C = c) \tag{3.1}$$

Analizziamo ora la probabilità condizionale di One-time-pad considerando le stringhe m e c come sequenze binarie lunghe n bit. Applicando la definizione di probabilità condizionale possiamo riscrivere la formula

$$\mathcal{P}(M = m|C = c) = \frac{\mathcal{P}(M = m, C = c)}{\mathcal{P}(C = c)} \tag{3.2}$$

Fissato un messaggio m chiavi diverse danno origine a crittogrammi diversi generati con probabilità $(1/2)^n$. Dunque, la probabilità $\mathcal{P}(C = c) = (1/2)^n$ per ogni c. Ciò significa che gli eventi $\{M = m\}$ e $\{C = c\}$ sono indipendenti. La probabilità congiunta di due eventi indipendenti non è altro che il prodotto delle probabilità. Pertanto

$$\frac{\mathcal{P}(M = m, C = c)}{\mathcal{P}(C = c)} = \frac{\mathcal{P}(C = c) \times \mathcal{P}(M = m)}{\mathcal{P}(C = c)} = \mathcal{P}(M = m) \tag{3.3}$$

I calcoli ci mostrano che One-time-pad è un cifrario perfetto. Bisogna considerare però i grossi limiti che il cifrario ci pone davanti

- le chiavi devono essere casuali e lunghe almeno quanto il messaggio
- le chiave devono essere usate solo una volta (in inglese one-time). Ripetendo la cifratura ti testi diversi con la stessa chiave si possono effettuare degli attacchi basati sulla probabilità che certe sequenze di lettere compaiano sul un testo.

Nonostante i problemi di one-time-pad, il cifrario non è scomparso ma si è evoluto. Negli algoritmi di tutti i cifrari simmetrici moderni, fra tanti cicli e permutazioni di bit, c'è almeno un passaggio nel quale si calcola l'or esclusivo

bit a bit tra un derivato della chiave e un derivato del messaggio. Qui di seguito riportiamo solo uno, forse il più evoluto e sicuro, dei cifrari simmetrici che siano stati progettati.

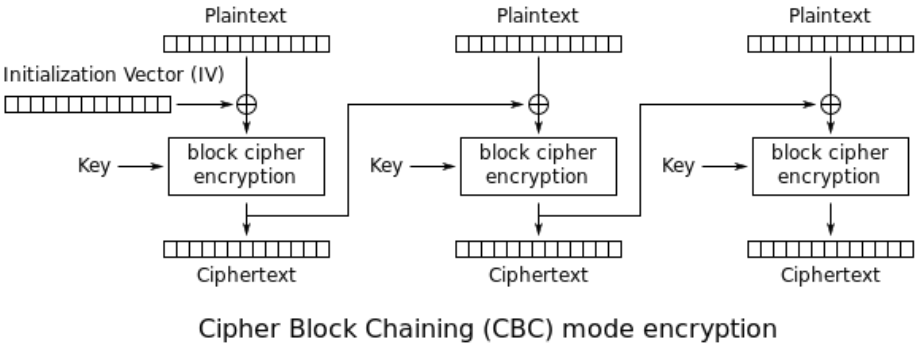
3.1.1 AES

L'Advanced Encryption Standard (AES), essendo riconosciuto dall'NSA come un cifrario top secret", è oggi uno degli algoritmi di crittografia più utilizzati e sicuri tra quelli disponibili. Nasce nel 1997, quando il National Institute of Standards and Technology (NIST) ha annunciato la ricerca di un successore per un vulnerabile DES. Dopo quasi 5 anni di ricerche, è stato standardizzato nel Novembre 2001, tra le varie proposte, l'algoritmo dei crittografi belga Daemen e Rijmen denominato Rijndael. La scelta è stata dettata dal fatto che l'algoritmo proposto eccelleva in sicurezza, prestazioni e flessibilità. L'algoritmo di default utilizza chiavi di sessione lunghe 128 bit e si basa su diverse sostituzioni non lineari, permutazioni e trasformazioni eseguite su blocchi di dati di 16 byte(128 bit). Tali operazioni vengono ripetute più volte, chiamate round. Sulla base di questa struttura a blocchi di AES, il cambiamento di un singolo bit sia nella chiave sia nei blocchi di testo, comporta un testo cifrato completamente diverso. Esistono inoltre più versioni del protocollo AES che si differenziano dalla lunghezza della chiave: 128, 192 o 256 bit. Tutti drastici miglioramenti rispetto alla chiave di 56 bit DES. A titolo di esempio: la decifrazione di una chiave AES a 128 bit dal cuore di un supercomputer richiederebbe un tempo più lungo della presunta età dell'universo. Ad oggi, non esiste nessun attacco praticabile contro AES. Pertanto, AES rimane lo standard di crittografia preferito da governi, banche e sistemi di alta sicurezza in tutto il mondo.

3.1.2 Cifrari a composizione di blocchi

I cifrari simmetrici a blocchi sono caratterizzati dal fatto che due blocchi uguali comportano la stessa sequenza cifrata. Per questo motivo i cifrari a blocchi so-

Figura 3.1: Schema mostrante il funzionamento della tecnica del CBC



no soggetti a determinati attacchi che sfruttano sequenze composte da blocchi uguali per riuscire a derivare la chiave di cifratura. Sono stati proposti molti metodi per ovviare al problema ma quello più diffuso è certamente il *Cipher Block Chaining (CBC)*. Il metodo si basa sulla composizione del il testo cifrato del blocco i con il testo in chiaro del blocco $i+1$. Questa tecnica, nonostante la semplicità, riesce ad eliminare la periodicità dei messaggi rendendo il cifrario decisamente più robusto.

Anche utilizzando il CBC, il problema dell'attacco alla chiave potrebbe presentarsi nuovamente cifrando due sequenze identiche. Per questo motivo, come possiamo vedere nella figura 3.1, anche il primo blocco che si va a cifrare è composto con un vettore di inizializzazione.

Mettendo insieme tutte le componenti visti finora, per stabilire una comunicazione sicura, le due parti si devono accordare sul protocollo di cifratura (DES, AES-128, AES-192, AES-256, ecc.), su una chiave di sessione, e su un vettore di inizializzazione. Normalmente una delle due parti di una comunicazione decide protocollo, vettore di inizializzazione e chiave di sessione e le comunica all'altro. Mentre il protocollo e il vettore di inizializzazione possono essere inviati in chiaro, la chiave di sessione deve essere in qualche modo cifrata affinché solo il destinatario riesca a decifrarla. Si ha quindi la necessità di introdurre dei cifrari asimmetrici i cui utilizzatori dovranno riuscire ad inviare informazioni nonostante il possesso di chiavi diverse.

3.2 Cifratura asimmetrica

Nel 1976 Diffie e Hellman, e indipendentemente Merkle, introdussero un concetto che avrebbe rivoluzionato tutte le comunicazioni segrete: i cifrari a chiave pubblica. Mentre nei cifrari simmetrici la chiave di cifratura è la stessa per la decifrazione ed è nota solo ai due partner, nei cifrari a chiave pubblica, o asimmetrici, le chiavi di cifratura e decifrazione sono completamente diverse tra loro. Esse sono scelte dal destinatario che rende pubblica la chiave di cifratura mentre mantiene segreta la chiave di decifrazione.
Definiamo ora:

- k_U^+ la chiave pubblica dell'utente U
- k_U^- la chiave privata dell'utente U
- $C(m, k)$ la funzione di cifratura
- $D(m, k)$ la funzione ci decifratura

Il protocollo di cifratura asimmetrica dovrà garantire:

- $\forall m : D(C(m, k_U^+), k_U^-) = m$
- $< k_U^+, k_U^- >$ deve essere facile da calcolare e deve risultare praticamente impossibile che due utenti scelgano la stessa coppia di chiavi
- dati m e k_U^+, è facile per il mittente calcolare calcolare $c = C(m, k_U^+)$
- dati c e k_U^-, è facile per il destinatario calcolare $m = D(c, k_U^-)$
- pur conoscendo c, k_U^+ e le funzioni C e D, è difficile per il crittoanalista risalire al messaggio m e alla chiave privata k_U^-.

I termini "facile" e "difficile", come in tutto il resto del capitolo, si riferiscono al costo computazionale impiegato per risolvere un determinato problema.

3.2.1 RSA

RSA è un algoritmo di cifratura a chiave pubblica inventato nel 1978 da Rivest, Shamir e Aldeman. L'algoritmo da loro inventato, denominato RSA per via delle iniziali dei loro cognomi, non è sicuro da un punto di vista matematico teorico ma è molto affidabile data l'enorme mole di calcoli e l'enorme dispendio di tempo necessari per decifrare il messaggio.

Il funzionamento si basa sul fatto che, con la tecnologia attuale, l'operazione di moltiplicazione è facile mentre non si può dire altrettanto per l'operazione di fattorizzazione. La differenza tra i costi computazionali delle due operazioni cresce esponenzialmente con l'aumentare della lunghezza della chiave.

Il protocollo RSA, come tutti i protocolli a chiave pubblica, è composto da tre fasi.

Creazione della chiave Il destinatario di un messaggio genera una coppia di chiavi attraverso le seguenti operazioni:

1. Scegliere due numeri primi molto grandi (300 bit): p e q
2. Calcolare $n = p \times q$, e la funzione di Eulero $\Phi(n) = (p-1)(q-1)$
3. Scegliere un intero e minore di $\Phi(n)$ e primo con esso.
4. Calcolare l'intero d inverso di e modulo $\Phi(n)$ (l'esistenza ed unicità di d è garantita dal fatto che e e $\Phi(n)$ sono primi tra loro).
5. Rendere pubblica la chiave $k_U^+ =< e, n >$ e mantenere segreta la chiave $k_U^- =< d >$

Cifratura Il messaggio è codificato come sequenza binaria e trattato come un numero intero m. m deve essere minore di n e per garantire ciò si divide il messaggio in blocchi di lunghezza $(log_2 n) - 1$. L'operazione di cifratura consiste nel calcolo del crittogramma

$$c = C(m, k_U^+) = m^e \bmod n \tag{3.4}$$

Decifratura Il crittogramma c viene decifrato calcolando

$$m = D(c, k_U^-) = c^d \bmod n \tag{3.5}$$

La correttezza del cifrario è garantita dal fatto che:

$$m = D(C(m, k_U^+), k_U^-) = \tag{3.6}$$

Sostituendo la formula di cifratura 3.4

$$= D(m^e \bmod n, k_U^-) = \tag{3.7}$$

Sostituendo la formula di decifratura 3.5

$$= (m^e \bmod n)^d \bmod n = \tag{3.8}$$

$$= (m^e)^d \bmod n = \tag{3.9}$$

$$= m^{e \times d} \bmod n = \tag{3.10}$$

Per costruzione di d: $d \times e = 1 \bmod \Phi(n)$

$$= m \times m^{\Phi(n)} \bmod n = \tag{3.11}$$

$$= (m \bmod n) \times (m^{\Phi(n)} \bmod n) = \tag{3.12}$$

Per il teorema di Eulero:$m^{\Phi(n)} = 1 \bmod n$

$$= m \bmod n = \tag{3.13}$$

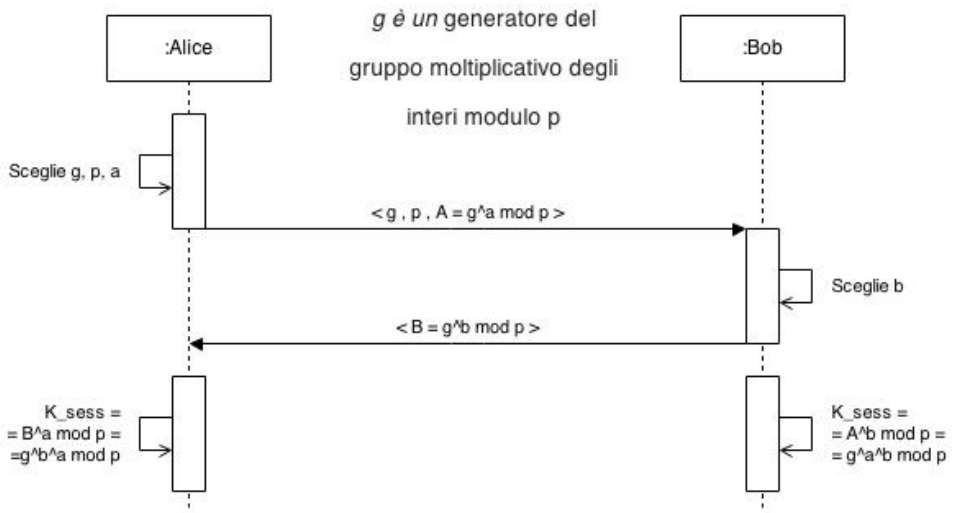
Per definizione di m: $m < n$

$$= m \bmod n = m \tag{3.14}$$

Come si può vedere, i calcoli del protocollo RSA sono commutativi. Grazie a questa proprietà si può cifrare o decifrare un messaggio sia con la chiave pubblica, $< e, n >$, sia con quella privata, $< d, n >$. Questa proprietà è molto utile per verificare che un determinato messaggio è stato cifrato da una determinata persona. Infatti se si cifra un messaggio con una chiave privata di una persona P, questo può essere decifrato solo con la chiave pubblica di P. Infatti, i sistemi di firma digitale si basano proprio su questa caratteristica del protocollo RSA.

Il problema del protocollo descritto è la poca velocità delle operazioni di creazione della chiave, cifratura e decifratura. Non è conveniente utilizzare

Figura 3.2: Funzionamento standard del protocollo di Diffie-Hellman



questo protocollo per la cifratura di una comunicazione e per questo motivo, si utilizzano degli approcci ibridi: si utilizza RSA o un qualsiasi protocollo a chiave pubblica per inviare al destinatario una chiave di sessione dopodiché si cifrano i messaggi successivi con protocolli simmetrici come AES.

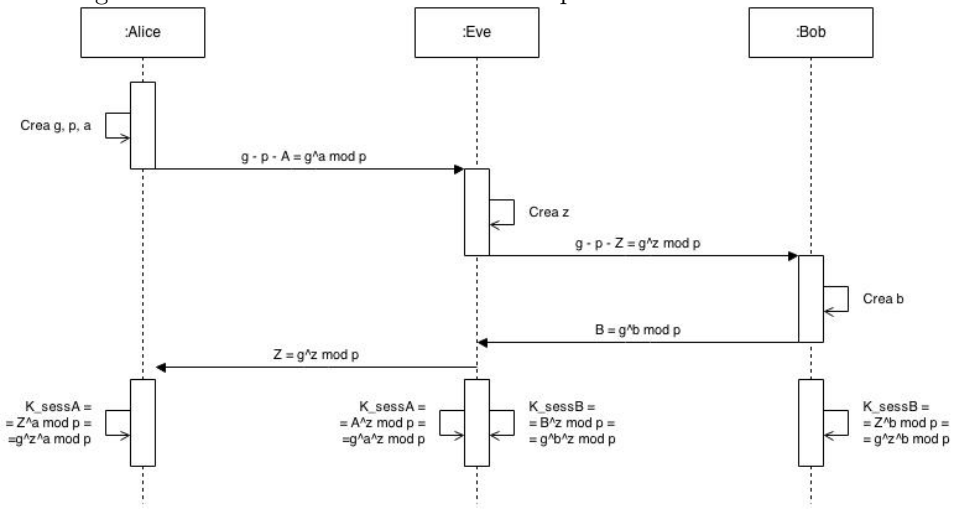
3.2.2 Diffie-Hellman

Nel 1976, Whitfield Diffie e Martin Hellman[21] progettarono un protocollo che consentisse a due parti che vogliono stabilire una comunicazione di accordarsi su una chiave di sessione segreta utilizzando un canale di comunicazione insicuro. Lo scambio di chiavi Diffie-Hellman è un protocollo che non richiede che le parti si siano precedentemente scambiate delle informazioni o si siano incontrate in precedenza. La chiave ottenuta può essere utilizzata come chiave di un cifrario simmetrico attraverso il quale crittografare tutti i messaggi.

L'idea che sta a monte del protocollo di Diffie-Hellman si basa sul fatto che è computazionalmente facile calcolare una funzione esponenziale mentre è molto difficile calcolare il logaritmo discreto.

Definiamo in un primo momento un generatore modulo n. Questo è un intero le cui potenze modulo n sono congruenti con i numeri coprimi ad n. Sup-

Figura 3.3: Attacco man-in-the-middle al protocollo di Diffie-Hellman



poniamo ora che Alice e Bob decidano di stabilire una comunicazione. Come si può vedere in figura 3.2. Nell'esempio mostrato spetta ad Alice cominciare la connessione verso Bob generando un numero primo e g un generatore del gruppo moltiplicativo degli interi modulo p. Oltre a questo, Alice genera un numero casuale a che sarà la propria chiave privata e calcola $A = g^a \bmod p$ che diventerà la propria chiave pubblica. Ad Alice non resta altro che inviare i parametri p e g e la chiave pubblica A a Bob il quale, una volta ricevuti i messaggi, genererà un numero casuale b e calcolerà la propria chiave pubblica $B = g^b \bmod p$ che invierà ad Alice. Ci troviamo in un punto in cui sia Alice che Bob conoscono le proprie chiavi private e le chiavi pubbliche dell'altro interlocutore. Non resta altro che generare una chiave di sessione per cifrare i messaggi successivi. Questa chiave viene calcolata semplicemente calcolando la chiave pubblica ricevuta elevato alla propria chiave privata. Alice, quindi, calcolerà $K = B^a = (g^b)^a = g^{a \times b}$ e mentre Bob calcolerà $K = A^b = (g^a)^b = g^{a \times b}$.

Purtroppo l'algoritmo Diffie-Hellman, come altri protocolli asimmetrici, è vulnerabile all'attacco "Man in the middle". L'attacco è effettuato da un terzo il quale riesce ad intercettare i messaggi tra Alice e Bob posizionandosi "nel mezzo" tra i due. L'obiettivo del terzo che chiameremo Eve, è quello di falsificare le chiavi pubbliche di Alice e Bob ed ingannare le due parti. Infatti,

per come è stato descritto finora nessuno garantisce l'autenticità delle chiavi pubbliche ricevute dall'altro interlocutore. L'esempio in figura 3.3 mostra come Eve si frappone tra Alice e Bob generando una propria chiave pubblica e generando due chiavi di sessione, una con Alice e l'altra con Bob. Se Eve inoltra correttamente i messaggi al reale destinatario, i nostri amici non sapranno mai che la loro conversazione è stata intercettata e letta da una terza parte.

Per evitare l'attacco è necessario che le chiavi pubbliche inviate all'altro interlocutore non siano falsificabili. Questo obiettivo è attuabile utilizzando i servizi di un'altra entità fidata la quale si occuperà di firmare le chiavi pubbliche generate prima di inviarle all'interlocutore.

3.3 La firma digitale

Prima di parlare del funzionamento della firma digitale è bene analizzare quali ragioni rendono necessaria una firma sia digitale che manuale. La firma si aggiunge ad un documento per provare l'autenticità e la paternità del documento stesso oppure per siglare un accordo raggiunto. Essa è autentica, non falsificabile, non riutilizzabile e non può essere ripudiata da chi l'ha apposta.

Autentica e non falsificabile significa che chi l'ha apposta è veramente colui che sottoscrive il documento.

La firma non è riutilizzabile significa che la firma è strettamente legata al documento su cui è apposta.

Il documento non è alterabile quindi la firma è legata al documento nella sua forma originale.

La Firma non può essere ripudiata e ciò garantisce una prova legale di un accordo o di quanto descritto in un documento.

Per implementare un sistema di firma digitale abbiamo bisogno di un sistema crittografico che permetta ad ogni utilizzatore di mantenere qualcosa di privato: stiamo parlando della crittografia asimmetrica. In particolar modo,

abbiamo visto che il protocollo RSA può essere utilizzato per cifrare o decifrare utilizzando sia la chiave pubblica che quella privata. Grazie a questa proprietà, se un documento si cifra con la chiave privata di un utente, si può decifrare solo con la chiave pubblica dello stesso utente. Se decifratura con la chiave pubblica va a buon fine, significa che il documento è stato emesso dal proprietario della chiave pubblica utilizzata.

Vediamo ora come è possibile implementare la firma digitale utilizzando il protocollo RSA. Innanzi tutto abbiamo bisogno di funzioni hash crittografiche come MD5 o SHA-1. Queste funzioni sono facili da calcolare ma sono difficili da invertire. Ovvero, dato il risultato di una funzione hash, non è computazionalmente possibile trovare una stringa dalla quale la funzione è stata calcolata a meno di provare tutte le possibili stringhe: soluzione che, a seconda della lunghezza della hash calcolata, potrebbe impiegare anche milioni di anni di calcolo.

Alleghiamo ora al documento un *Message Authentication Code (MAC)* ottenuto calcolando l'hash del documento stesso. Questo MAC **non è riutilizzabile** in quanto è strettamente dipendente dal documento su cui è stata calcolata e ci garantisce che il documento **non è stato alterato** in quanto non è computazionalmente possibile creare un nuovo documento tale che la sua funzione hash sia uguale al MAC allegato. In questo modo, chi riceve un documento, non deve far altro che calcolare la funzione hash del documento stesso e confrontarla con il MAC allegato.

Per avere una firma digitale basta cifrare il MAC con la chiave privata di chi emette il documento. In questo modo, non solo garantiamo che il documento non è stato alterato, ma garantiamo anche che il creatore del documento è colui al quale appartiene la chiave pubblica utilizzata per la decifratura del MAC.

Il protocollo di firma digitale, sfruttando un protocollo asimmetrico, è soggetto all'attacco Man-In-The-Middle in quanto un attaccante può falsificare la chiave pubblica e quindi la firma. Come accennato nel paragrafo precedente, in questi casi si ha bisogno di chiamare in causa una terza parte fidata che cer-

tifichi l'associazione tra una determinata persona o azienda e la relativa chiave pubblica. Questa terza entità possiede anch'essa una coppia di chiavi RSA che utilizza per firmare un certificato nel quale, tra le altre cose, è segnalata l'associazione tra un utente e la propria chiave pubblica. Questa entità viene appunto chiamata *Certification Authority (CA)*.

Si potrebbe pensare che la vulnerabilità è semplicemente spostata di un livello: dall'utente finale alla CA. Questo è vero solo in parte in quanto esistono poche CA nel mondo ed è quindi semplice installare e mantenere nel sistema un numero ristretto di chiavi pubbliche.

3.4 La dimostrazione Zero-Knowledge

L'autenticazione Zero-Knowledge[9] è un tipo di autenticazione nel quale si vuole dimostrare di essere a conoscenza di un'informazione senza rivelarla all'interlocutore. Per capire meglio il concetto, spieghiamolo con un esempio. Ipotizziamo di avere due oggetti completamente identici ma di colore diverso. Ipotizziamo inoltre di voler dimostrare ad un nostro amico che gli oggetti in questione hanno un colore diverso senza però rivelargli i colori. La soluzione banale sarebbe quella di mostrargli entrambi gli oggetti così che l'amico stesso possa constatare il diverso colore ma, in tal caso, verrebbe a conoscenza dei colori che vogliamo mantenere segreti. Una soluzione più elaborata consiste nel bendare il nostro amico, dargli entrambi gli oggetti, uno per mano, e chiedergli di nasconderli dietro la schiena. La dimostrazione avviene per fasi. Ad ogni fase, il nostro amico può scegliere di scambiare gli oggetti che ha in mano senza dirci se ha effettuato o no lo scambio. Quindi, ci mostrerà solo un oggetto. Grazie ai diversi colori, siamo in grado di capire, e quindi indovinare, se il nostro amico ha invertito gli oggetti in mano.

Il nostro amico, che non è convinto che gli oggetti abbiano colore diverso, può pensare che l'aver indovinato la soluzione giusta sia un caso dovuto alla fortuna. Infatti al primo tentativo la probabilità di indovinare la soluzione giusta è di 1/2. Andando avanti con la dimostrazione, la probabilità di indovi-

nare tutte le volte se è stato effettuato uno scambio di due oggetti dello stesso colore diventa talmente bassa che, il nostro amico, si convince del contrario ovvero che gli oggetti in suo possesso hanno un colore diverso.

La soluzione appena proposta non è altro che una dimostrazione Zero-Knowledge. Infatti siamo riusciti ad dimostrare un concetto senza però rivelare nessuna informazione. Una dimostrazione a conoscenza zero deve soddisfare tre proprietà:

Completezza : se l'affermazione è vera, un dimostratore onesto potrà convincere del fatto un verificatore onesto.

Correttezza : se l'affermazione è falsa, nessun dimostratore imbroglione potrà convincere il verificatore onesto che essa è vera, o meglio la probabilità di riuscire a convincerlo può essere resa bassa a piacere.

Conoscenza zero : se l'affermazione è vera, nessun verificatore imbroglione potrà sapere altro che tale informazione.

La tecnica delle dimostrazioni a conoscenza zero può essere utilizzata per l'autenticazione di un utente. Il fatto da dimostrare consiste nel possesso di una password senza inviare direttamente tale password attraverso un canale non sicuro. In questo caso specifico, sia il dimostratore che il verificatore sono a conoscenza della password da verificare. Il protocollo consiste in una serie di prove attraverso il quale:

1. il verificatore invia una stringa casuale chiamata nonce.
2. il dimostratore cifra la stringa con un cifrario simmetrico utilizzando come chiave la password da verificare.
3. il verificatore decifra la stringa con la password di sua conoscenza e controlla se il nonce risultante è uguale a quello invato precedentemente.

Quando il verificatore è soddisfatto, può interrompere il susseguirsi di prove e autenticare l'utente. Grazie a questo protocollo anche il dimostratore che non è sicuro sull'identità del verificatore può autenticarsi senza la paura di rivelare la password ad un verificatore corrotto.

Capitolo 4

Izzie: l'architettura

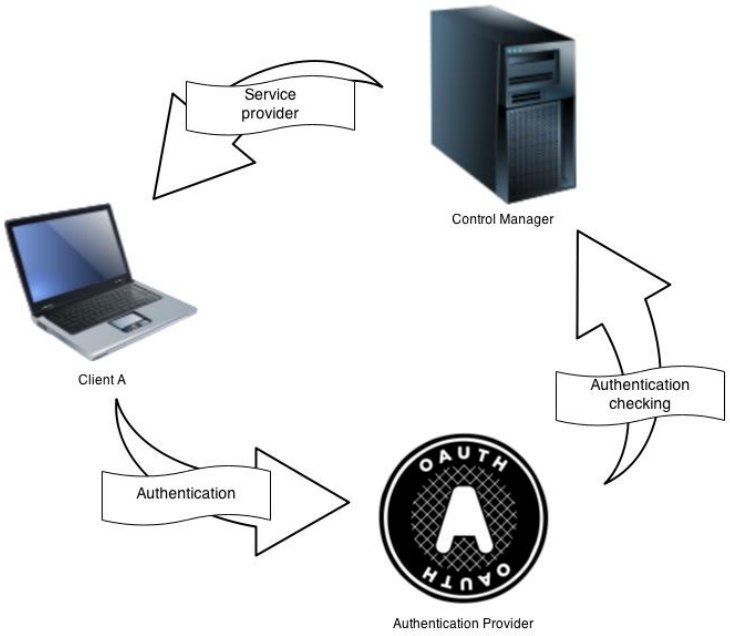
Izzie è un'infrastruttura progettata per unire i benefici delle architetture cloud e P2P dove gli utenti possono usufruire di più servizi in totale sicurezza. L'architettura è suddivisa in tre blocchi con compiti distinti: client, autenticazione e controllo. Il sistema, per funzionare correttamente, ha la necessità di almeno un'entità appartenente a ciascun blocco.

La piattaforma Izzie è stata progettata per essere il più generale possibile e, quindi, poter ospitare una vasta gamma di servizi che sono gestiti da un controllo implementato ad-hoc. Ogni servizio richiederà che gli utenti che lo utilizzano siano autenticati attraverso qualche altro servizio di autenticazione che può essere anche esterno al servizio stesso.

Il client può interagire indifferentemente con istanze diverse del blocco controllo grazie alle comunicazioni realizzate attraverso chiamate REST[16] e messaggi JSON[13]. Descriviamo ora con maggiore dettaglio i blocchi che compongono l'architettura Izzie.

L'autenticazione è il blocco che si occupa della fase di login e garantisce l'avvenuta autenticazione dell'utente. Tutti i servizi che implementano il protocollo OAuth[20] sono potenziali istanze di questo blocco. Il protocollo OAuth è già implementato e utilizzato con ottimi risultati dalla maggior parte dei social network ed è possibile integrarlo come blocco di autenticazione dei servizi Izzie.

Figura 4.1: Schema ad alto livello dell'architettura presentata



Il controllo è il blocco che si occupa di gestire le funzionalità che sono necessarie per mantenere attivi uno o più servizi. Ogni istanza del blocco Controllo implementa un'interfaccia HTTP conforme con la filosofia REST. Una volta definita l'interfaccia del servizio, ognuno può implementarsi un controllo per gestire il proprio servizio.

Il client si trova al centro di tutto il sistema in quanto è l'unico blocco fisso. Si occupa di far accedere un utente ad ogni servizio al quale l'utente è registrato interagendo con una delle istanze del blocco di autenticazione supportate dal servizio stesso. A questo punto, insieme al supporto del controllo, il client implementa le funzionalità dei servizi e l'interfaccia per utilizzarli. Una novità di Izzie rispetto alle normali architetture client/server, sta nel fatto che sul client vengono eseguite una parte delle funzionalità tra cui quelle riguardanti la gestione delle chiavi. Inoltre, il client è il blocco che implementa il protocollo P2P e per questo verrà descritto molto più dettagliatamente nei capitoli successivi.

La separazione della business logic in diversi blocchi consente la personalizza-

zione dell'architettura e permette di creare un servizio gestito da un controllo ad-hoc che richieda l'autenticazione presso un determinato server OAuth. Per esempio, è possibile per un'azienda implementare un sistema di storage interno che richieda l'autenticazione attraverso i server di Google. Il controllo del servizio potrebbe risiedere su un dispositivo di storage(NAS) installato nella rete locale all'interno delle mura aziendali.

In questo capitolo analizzeremo in un primo momento l'approccio Rest per poi concentrarsi sulle funzionalità dei blocchi. L'autenticazione e controllo non verranno approfonditi in dettaglio perché, anche se fanno parte dell'architettura, sono state sviluppate in un'altra tesi. Il client, invece, verrà discusso in maniera più dettagliata in quanto, insieme alla DHT progettata, è l'argomento principale di questa tesi. In questo capitolo verrà dedicato un paragrafo per spiegare le funzionalità del client e come questo si interfaccia con il resto dell'infrastruttura mentre nel prossimo capitolo verrà spiegato il protocollo P2P progettato.

4.1 L'approccio REST

La definizione di REST[16] è apparsa per la prima volta nel 2000 nella tesi di dottorato di Roy Fielding, "Architectural Styles and the Design of Network-based Software Architectures", discussa presso l'Università della California, ad Irvine. In questa tesi si analizzavano alcuni principi alla base di diverse architetture software, tra cui appunto i principi per la definizione di un'architettura software che consentisse di vedere il Web come una piattaforma per l'elaborazione distribuita.

REST è una filosofia di programmazione in quanto non si riferisce ad un sistema concreto e ben definito nè si tratta di uno standard stabilito da un organismo di standardizzazione. Si definisce invece, un insieme di principi architetturali per la progettazione di un sistema elencati. Tali principi sono:

1. Lo stato e le funzionalità dell'applicazione sono formati da risorse
2. Ogni risorsa è unica e indirizzabile usando sintassi universale

3. Tutte le risorse sono disponibili attraverso un'interfaccia uniforme
4. Ogni risorsa è caratterizzata da:
    - un insieme vincolato di operazioni ben definite
    - un insieme vincolato di contenuti,

L'approccio REST definisce anche i seguenti vincoli applicati ad una architettura:

Client–server . Client e Server sono separati da interfacce. In questo modo si ha una netta separazione dei ruoli. In questo modo Server e client possono essere sostituiti e/o sviluppati in parallelo fintanto che l'interfaccia non viene modificata.

Stateless . Durante la comunicazione tra client e server ogni richiesta è indipendente dalle altre. Il client mantiene lo stato della propria sessione oppure mantiene un identificatore dello stato stesso che verrà allegato ad ogni richiesta inviata. Il server può mantenere delle informazioni sullo stato del client solamente attraverso un servizi esterni come il database ai quali si accede attraverso l'identificatore che si riceve dal client stesso.

Cacheable . I client possono fare caching delle risposte le quali devono definirsi implicitamente o esplicitamente cacheable o no, in modo da prevenire che i client possano riusare stati vecchi o dati errati. Una gestione ben fatta della cache può ridurre le comunicazioni client server, migliorando scalabilità e performance.

Code on demand . I server possono estendere o personalizzare le funzionalità del client trasferendo del codice eseguibile come Applet Java o linguaggi di scripting client side come JavaScript. L'utilizzo di questa funzione permette di semplificare la gestione degli aggiornamenti automatici mantenendo sempre il client aggiornato all'ultima versione rilasciata.

Bisogna notare che i principi REST sono completamente astratti quindi, parlando di REST non si parla necessariamente di Web. Il World Wide Web,

infatti, risulta essere un'istanza delle applicazioni dei principi REST che si presta bene all'implementazione di queste architetture in quanto è contraddistinto dalle caratteristiche elencate di seguito.

- Identificazione delle risorse attraverso URI definite
- Utilizzo esplicito dei metodi HTTP
- Risorse autodescrittive
- Collegamenti tra risorse
- Comunicazione senza stato

Inizialmente questa visione del Web è stata un po' trascurata, ma negli ultimi anni l'approccio REST è venuto alla ribalta come metodo per la realizzazione di Web Service altamente efficienti e scalabili ed ha al suo attivo un significativo numero di applicazioni. Come riscontro pratico di quanto detto, si può notare come, nell'ultimo anno, molti servizi abbiano cominciato a convertire le proprie API seguendo la filosofia Rest.

Avendo terminato di analizzare il formato json e la filosofia rest, abbiamo tutti i componenti per poter cominciare ad analizzare il funzionamento dei blocchi descritti in precedenza.

4.2 Il client

Il client è la parte centrale di tutta l'infrastruttura Izzie, il suo compito è quello di far autenticare l'utente con gli authentication provider supportati dai controlli dei servizi al quale l'utente è iscritto, interagire con i controlli stessi e, se i servizi che richiedono un supporto P2P, il client dovrà implementare il protocollo distribuito che lo renderà parte integrante della DHT che analizzeremo in dettaglio nel prossimo cAPItolo.

Durante tutte le comunicazioni tra peer, si ha la necessità che nessuno riesca ad intercettare e leggere i messaggi scambiati. Per questo motivo tutte le chiavi di sessione devono essere generate dai client e inviate in maniera sicura all'altro

interlocutore. Stando a quanto detto nel cAPItolo precedente, in questo caso c'è bisogno di introdurre una coppia di chiavi asimmetriche per cifrare la chiave di sessione in modo tale che possa essere decifrata solo dal destinatario voluto.

Un'altra soluzione è quella che prevede la generazione della chiave di sessione in maniera costruttiva con l'apporto di entrambi i peer interessati nella comunicazione attraverso il protocollo di Diffie-Hellman. Questo protocollo, però, ha un nota vulnerabilità nei cofronti dell'attacco man-in-the-middle. Per evitare ciò, l'unico modo è quello di evitare la contraffazione delle chiavi pubbliche firmandole con un'altra coppia di chiavi asimmetriche di tipo RSA.

Entrambi gli approcci presentati per calcolare una chiave di sessione da utilizzare in tutte le comunicazione P2P richiedono l'introduzione di chiavi RSA. Nonostante il secondo approccio richieda l'utilizzo di un ulteriore protocollo (Diffie-Hellman), è stato preferito al primo perché il concetto di "generazione costruttiva della chiave" si sposa meglio con la filosofia di P2P.

Ovviamente anche le chiavi RSA devono essere generati dai client ma, essendo anch'esse chiavi asimmetriche, sono facilmente falsificabili e soggette quindi all'attacco un man-in-the-middle. Come accade attualmente sulla rete internet, il problema delle chiavi pubbliche non autentiche si risolve introducendo un'autorità fidata e riconosciuta da tutti che si occupa di firmare dei certificati contenenti, tra le altre cose, l'associazione tra la chiave pubblica e l'identità dell'utente. Nella rete Izzie, questo compito spetta al controllo il quale è responsabile dell'identificazione dell'utente e della firma dei certificati.

Tutte le informazioni salvate sulla DHT di cui il client fa parte, sono firmate dagli utenti quindi, per ogni accesso alla rete è necessario utilizzare sempre le stesse chiavi RSA. Finché l'utente si connette sempre dallo stesso client il problema si risolve semplicemente memorizzando la coppia di chiavi sul dispositivo stesso. Il problema sorge quando ci si vuole connettere da più dispositivi, anche contemporaneamente, ed avere sempre a disposizione gli stessi dati con gli stessi diritti. La soluzione progettata e implementata nel sistema Izzie consiste nell'introduzione di una "MasterKey" con la quale si cifra la coppia di chiavi RSA così da poterla memorizzare sul controllo ed ottenerla ad ogni accesso.

Per rendere più usabile il sistema, non si richiede che l'utente memorizzi un'intera chiave di cifratura ma semplicemente una "MasterPassword". La "MasterKey" può essere calcolata partendo dalla "MasterPassword" attraverso un calcolo di una serie di funzioni hash.

4.3 Il blocco Autenticazione

Il blocco Autenticazione si occupa di gestire direttamente i dati dell'utente permettendo l'operazione di Login. Data la flessibilità dell'architettura Izzie, il blocco di autenticazione può essere qualsiasi servizio che implementa il protocollo OAuth 2.0.

Open Authorization o più comunemente OAuth, è un protocollo di comunicazione open mediante il quale un servizio di autenticazione può gestire in modo sicuro l'accesso da parte di terzi ai dati degli utenti, proteggendo contemporaneamente le loro credenziali.

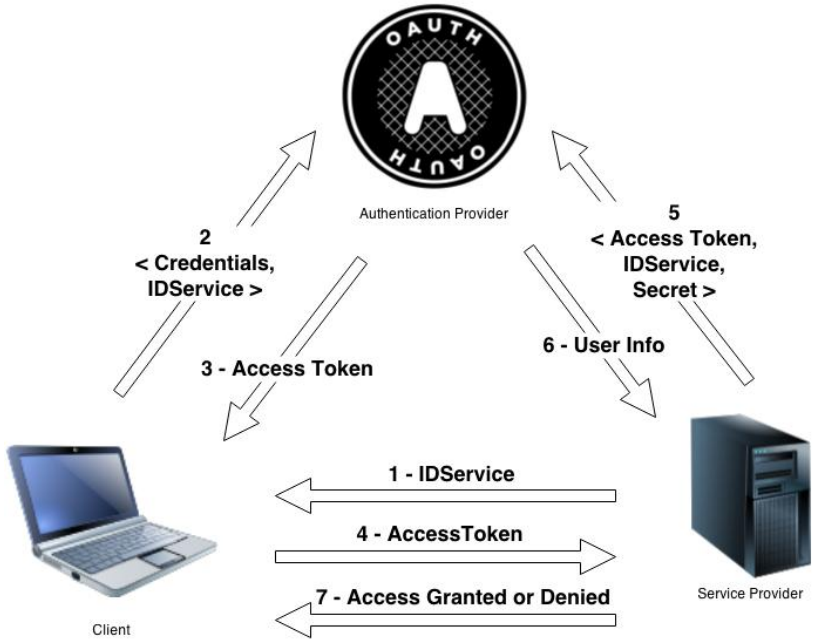
Il protocollo OAuth è nato con l'obiettivo di proporsi come alternativa aperta ai molti protocolli proprietari già esistenti, come Google AuthSub[24], Yahoo BBAuth[25] e tanti altri. OAuth, dalla versione 1.0[19], recependo via via le RFC proposte dai vari esperti, è in continua evoluzione e in questo momento è in uso la versione 2.0[20].

Il protocollo, durate la sua esecuzione, vede coinvolti tre attori:

Authentication provider è il servizio attraverso il quale l'utente si registra e si autentica(es. Facebook, Google+, Twitter, ecc). Questo è quindi in grado di fornire le informazioni sull'utente a chi dimostra di essere autorizzato dall'utente stesso.

Service provider è la applicazione che, avendo ricevuto il permesso dell'utente, richiede alcuni dati dello stesso. Nel nostro caso specifico il service provider è il controllo che richiede l'identità dell'utente. L'applicazione deve essere registrata presso l'Authentication provider. Con questa operazione, l'Authentication Provider fornisce al Service provider le chiavi

Figura 4.2: Schema del funzionamento del protocollo OAuth



segrete che verranno utilizzate per lo scambio dei dati: IDService e Secret. Il primo è l'ID che l'Authentication provider assegna al Service provider mentre il secondo è un token segreto generato dall'Authentication provider che deve conoscere solo il Service provider.

L'utente è colui che, essendo registrato presso l'Authentication provider, vuole sfruttare i servizi offerti dal Service Provider evitando una nuova registrazione.

Il protocollo prevede le seguenti fasi che sono schematizzate in figura 4.2

1. L'applicazione redirige l'utente ad un URL predisposto dall'Authentication Provider per la fase di autenticazione e gli fornisce anche l'IDService che identifica il servizio e permette all'Authentication Provider di discriminare tra le applicazioni registrate.

2. L'utente accede alla pagina di autenticazione predisposta dall'Authentication Provider dove inserisce le proprie credenziali di accesso. Insieme alle credenziali viene inviato anche l'IDService in modo tale che l'Authenti-

catin Provider possa sapere su quale applicazione l'utente sta accedendo. Di solito la pagina di autenticazione lavora con il protocollo HTTPS il quale garantisce che i dati scambiati tra browser e web server siano codificati e non vengano trasferiti in chiaro come invece avviene nel protocollo HTTP. In questo caso la comunicazione risulta molto più sicura in caso di intercettazione.

3. L'Authentication Provider verifica l'identità dell'utente e, in caso di esito positivo, presenta all'utente la lista delle informazioni che il service provider ha richiesto. Per procedere l'utente deve approvare la richiesta concedendo così all'authentication provider il permesso di fornire al Service Provider specifico le informazioni elencate. L'Authentication Provider redirige così l'utente all'URL specificata nel punto 1. Nella richiesta di redirect è presente una stringa (detta token) che deve essere utilizzata successivamente dal Service Provider per richiedere eventuali informazioni all'Authentication Provider.

4. L'utente accede nuovamente alla pagina del Service Provider fornendo il token ricevuto al punto precedente.

5. Il Service Provider invia una richiesta HTTP all'Authentication Provider nella quale richiede i dati di cui ha bisogno specificando il token precedentemente ricevuto dall'utente, il proprio IDService e Secret ricevuti durante la fase di registrazione del servizio. L'Authentication provider utilizzerà l'IDService per discriminare tra le applicazioni registrate e confronterà il Secret ricevuto con quello generato durante la registrazione dell'applicazione. Se il Secret è lo stesso ci garantisce che la richiesta sia arrivata proprio dall'applicazione registrata.

6. L'Authentication provider, dopo aver controllato la corrispondenza tra IDService e Secret, controllerà la corrispondenza tra IDService e l'access token ricevuto. Nel caso in cui tutti i controlli vadano a buon fine si inviano al Service provider le informazioni richieste che possono variare per ogni Authentication provider. Per esempio, Facebook, espone i dati

anagrafici dell'utente oppure i suoi ultimi post in bacheca; LinkedIn espone i collegamenti di un utente oppure le sue iscrizioni ai gruppi; Google fornisce moltissime informazioni come ad esempio l'elenco dei contatti in rubrica.

7. Infine il Service provider invierà un messaggio al client concedendo o negando l'accesso ai servizi.

Il protocollo è compatibile, anche se con qualche variazione, con qualsiasi tipologia di applicazione: desktop, web e mobile. Dato che il client Izzie è un'applicazione desktop, il protocollo mostrato nel paragrafo precedente subisce una leggera modifica. Dopo l'avvenuta autenticazione, il blocco di autenticazione non redirige l'utente ad un URL specificata bensì ad una pagina contenente il token. Il client, può estrarre il token dalla pagina html ricevuta per poi inviarlo al Service Provider in modo da continuare l'esecuzione del protocollo. Un'Authentication Provider può essere utilizzato da una stessa applicazione attraverso più interfacce (es. Web e Desktop). Per fare questo il Service Provider deve registrare più servizi con diverse URL di redirezione specifici per l'interfaccia utente utilizzata. Nel caso in cui la redirezione non è supporta, si utilizza una stringa definita per specificare l'opzione all'Authentication Provider.

4.4 Il blocco Controllo

Il controllo è il blocco che si occupa di gestire un determinato servizio. Il suo funzionamento è strettamente dipendente dal tipo di servizio implementato e di conseguenza anche le API che vengono messe a disposizione. Per la corretta interoperabilità con tutti gli altri blocchi dell'infrastruttura, dobbiamo definire dei comandi REST uguali per tutti i servizi offerti dai vari controlli. Per semplicità di notazione scinderemo l'API del controllo in due parti: quella contenente le operazione standard uguali per tutti i Controlli (APIStandard); quella dipendente dal servizio (APIService).

4.4.1 Il Controllo del servizio

Il controllo di un servizio fornisce delle APIService specifiche per il servizio implementato. Per esempio, se volessimo implementare un servizio di storage, dovremmo mettere a disposizione, tra gli altri, dei comandi per inviare al server un determinato file (es. $urlServizio/api/putFile$) e per poterlo scaricare sul nostro personal computer (es. $urlServizio/api/getFile$).

Per eseguire determinate operazioni bisogna essere autorizzati e questo rende necessaria l'integrazione con uno o più Authentication provider. Il controllo dovrà quindi mettere a disposizione anche una serie di APIStandard. Tra le più importanti introduciamo un comando REST (urlServizio/info) per richiedere delle informazioni sul servizio (es. Tipo di servizio, lista degli Authentication provider supportati, eventuali URL accessori, ecc.). Un utente che vuole usufruire del servizio offerto dal Controllo, riceve le informazioni dello stesso attraverso la richiesta appena descritta ed esegue l'autenticazione OAuth con uno degli Authentication Provider supportati sia dal client che dal Controllo. Una volta ricevuto il token c'è bisogno di inviarlo al Controllo in modo che quest'ultimo possa verificare l'identità dell'utente. Per questo motivo tra le APIStandard offerte dal Controllo ci deve essere un comando che permette all'utente di inviare il proprio access token e ricevere dei metadati sull'API REST specifica del servizio (urlServizio/services). Utilizzando questa procedura il client ha tutte le informazioni necessarie per poter sfruttare i servizi offerti da un Controllo.

Tra le informazioni che un client riceve è specificato se il servizio offerto è supportato da un server o da una rete P2P. In quest'ultimo caso il Controllo dovrà implementare anche le funzioni per la gestione dei peer e la loro certificazione.

4.4.2 Il Controllo Izzie

Ogni utente che utilizza la piattaforma Izzie è registrato ad un determinato range di servizi, ognuno dei quali è gestito da un Controllo che supporta alcuni

Authentication Provider. Per rendere il sistema usabile, bisogna evitare che l'utente, tutte le volte che vuole utilizzare il servizio, debba cercare sul Web il punto di accesso del controllo e aggiungerlo alla piattaforma. Per questa ragione ci viene in aiuto uno speciale Controllo il cui servizio implementato è la gestione delle informazioni degli utenti tra cui la lista dei servizi al quale un utente è iscritto. In questo modo all'utente non resta altro che accedere, al Controllo principale, ottenere le URL dove raggiungere i controlli dei servizi, per accedere così al servizio desiderato. Infatti il mantenimento della lista dei servizi sul client è adibita proprio al controllo generale.

Per una maggiore usabilità è importante notare che le pagine per il login presentate dagli Authentication Provider inviano anche dei cookie. Salvando questi cookie sul proprio computer, l'utente può evitare, ogni qualvolta viene eseguita l'applicazione Izzie, la ripetizione della procedura di Login per tutti i servizi al quale si è iscritti.

Dato che, secondo la politica Izzie, tutti i dati che viaggiano sulla rete devono essere cifrati dai client con chiavi generate sui client stessi, anche le informazioni relative all'utente non devono rimanere in chiaro sul Controllo principale. Con questo accorgimento il Controllo principale diventa un semplice archivio di stringhe cifrate e tutta la gestione dei servizi viene demandata al client. Ciò significa che il client, dopo ogni modifica, si deve preoccupare di cifrare le informazioni in suo possesso e aggiornare il Controllo principale. Ricordiamo che ogni utente si definisce una propria MasterPassword utilizzata per la generazione di una chiave con la quale si cifra la coppia di chiavi RSA in possesso di ogni utente. La stessa chiave è anche utilizzata per la cifratura di queste informazioni da salvare sul Controllo principale.

Grazie a questi accorgimenti di sicurezza, è quindi possibile per il client salvare sul Controllo Izzie, non solo la lista dei servizi con i relativi punti di accesso, ma anche l'elenco dei cookie o password di autenticazione. Questa ottimizzazione migliora notevolmente l'usabilità del sistema in quanto, un utente, dopo essersi loggato per la prima volta ad un servizio, anche cambiando dispositivo avrà bisogno di eseguire un solo accesso all'infrastruttura Izzie per

avere a disposizione tutti i servizi al quale è iscritto.

L'utente, ricordando una sola password, riesce ad accedere, in completa sicurezza, a vari servizi evitando che nessuno di questi, nemmeno i server di Izzie, sappiano più informazioni di quanto non sia strettamente necessario per il funzionamento.

4.5 Un esempio concreto

In questo capitolo abbiamo descritto i componenti dell'architettura Izzie. Per capire meglio il funzionamento proviamo a vedere come tutti questi ingranaggi si compongono insieme per far girare il motore del sistema in esame. Ipotizziamo per esempio che si voglia costruire sopra Izzie un servizio di condivisione di contenuti a pagamento. Grazie alla sicurezza dell'infrastruttura siamo sicuri che i contenuti possano essere letti soltanto da chi ne abbia acquisito il diritto pagando una certa quota. Dato che sicuramente ci sono dei contenuti più scaricati di altri, il servizio può utilizzare la DHT sicura per implementare una cache distribuita nella quale salvare i contenuti più scaricati evitando così di accedere tutte le volte a servizio di cloud storage.

Per sviluppare un servizio come quello descritto, è necessario implementare un proprio controllore su un server che possa interagire con un servizio di cloud storage nel quale sono memorizzati i contenuti richiesti dagli utenti. Visto che il servizio è a pagamento, non tutti gli utenti possono scaricare i contenuti. Risulta quindi necessario che gli utenti siano autenticati e, solo gli aventi diritto possano scaricare il contenuto interessato. Per autenticare gli utenti si decide di ricorrere ad gestore delle autenticazioni privato ma si vuole offrire anche la possibilità agli utenti di eseguire il login con Facebook.

In base a quanto previsto dal protocollo OAuth 2.0 c'è la necessità di registrare il servizio di gestione dei contenuti come applicazione del servizio di autenticazione privato e quello di Facebook.

L'utente, dal canto suo, accederà in un primo momento al controllo Izzie dal quale verrà a sapere, tra gli altri, dell'esistenza del nuovo servizio implementato

e il link per accedere al relativo controllo. Attraverso quest'ultimo controllo si viene a conoscenza che il servizio richiede l'autenticazione con Facebook e con l'Authentication provider privato. L'utente deciderà quale servizio usare per il login ed eseguirà il protocollo OAuth fino a ricevere un messaggio di avvenuta autenticazione da parte del controllo. Questo messaggio contiene anche la lista delle API Rest fornite dal controllo. In questo modo al client non resta altro che usufruire dei servizi messi a disposizione dal controllo stesso.

Capitolo 5

La DHT sicura organizzata a gruppi

Progettare una DHT è già di per sé un compito abbastanza arduo perché bisogna tenere in considerazione tante problematiche dovute alla distribuzione delle risorse e all'elevato *churn rate*. Nel capitolo 2 abbiamo potuto constatare che, volendo aggiungere anche la componente sicurezza, abbiamo bisogno di rinunciare a qualche caratteristica tipica dei sistemi P2P puri come la completa distribuzione. Si era infatti arrivati alla conclusione che l'unico modo utile per combattere certi tipi di attacchi è quello di inserire un'entità centrale e fidata che certifichi l'identità dei nodi. La DHT che presentiamo in questo capitolo si ispira alla rete Pastry descritta nel capitolo 1 ma è pensata per fornire ad un utilizzatore un alto livello di sicurezza.

L'obiettivo è quello di sfruttare le componenti centralizzate e fidate dell'infrastruttura Izzie per implementare un protocollo distribuito nel quale tutte le comunicazioni sono cifrate e garantite dal Controllo Izzie. Come spiegato nel capitolo precedente, ogni client della rete possiede una coppia di chiavi RSA utilizzate per autenticare chiavi di Diffie-Hellman con le quali si generano le chiavi di sessione. Inoltre, se un servizio si avvale del supporto di una rete P2P, il relativo controllo fornisce al client un certificato che garantisce l'associazione tra le informazioni del nodo, l'identità dell'utilizzatore e la corrispondente

chiave pubblica.

Tutti i dati memorizzati in una DHT sono cifrati sia quando risiedono su un nodo sia quando vengono trasmessi ad altri nodi. Per garantire una maggiore sicurezza, alcuni servizi possono richiedere che alcuni dati siano salvati solamente tra i dispositivi dei facenti parte di un determinato gruppo e che nessun altro peer della rete debba entrare in possesso di informazioni che non è autorizzato a possedere. Ipotizziamo che un'azienda voglia mantenere dei documenti riservati su un servizio di storage distribuito solo tra i computer interni e quelli degli gli impiegati. Il sistema di storage deve poter salvare i file su una DHT limitata solo ai peer utilizzati dagli utenti che ne hanno i diritti creando così un gruppo formato da un sottoinsieme di nodi della DHT base. Ipotizzando ora che la stessa azienda di cui sopra voglia salvare altri file solo sui computer utilizzati dal gruppo di dirigenti, il peer di uno di questi dovrà poter gestire documenti riservati ai dirigenti, documenti aziendali e parti di file memorizzati sulla rete globale.

Con questa gestione dei gruppi, la DHT può essere considerata come organizzazione di diverse DHT che possono annidarsi e intersecarsi.

In questo capitolo, in un primo momento analizzeremo gli algoritmi epidemici che, nonostante la loro semplicità, risultano essere molto efficaci nella diffusione di informazioni e che, negli ultimi anni, stanno riscontrando particolare successo in campo della ricerca. Nel resto del capitolo si analizzeranno le principali caratteristiche e protocolli adottati dalla DHT Izzie al fine di limitare gli attacchi mostrati nel capitolo 2. Si vedranno infatti le caratteristiche generali di un nodo prima di parlare dell'organizzazione dello spazio delle chiavi, dislocazione delle risorse con il relativo protocollo di routing per accedervi. Analizzeremo, infine come vengono implementati e gestiti i gruppi.

5.1 Gli algoritmi epidemici

Gli algoritmi epidemici[26] sono stati pensati come tecnica semplice ed efficiente per gestire le repliche nei database distribuiti nei quali esiste un nodo

responsabile di una informazione che può effettuare gli aggiornamenti. Ad ogni aggiornamento è associato un timestamp utile per discriminare se la versione in nostro possesso è aggiornata oppure bisogna sostituire il dato con quello in possesso di un'altro nodo della rete. Questi algoritmi, per diffondere velocemente un dato, utilizzano un paradigma che simula fenomeni naturali come la diffusione dei virus o il pettegolezzo umano. L'idea di base è molto semplice: esiste una popolazione formata da un insieme di entità comunicanti con una visione locale dell'ambiente. Ogni entità può trovarsi in uno dei seguenti stati:

Infettabile : l'entità non è ancora venuta a conoscenza dell'informazione.

Infetta : l'entità è venuta a conoscenza dell'informazione e può diffonderla seguendo un insieme di regole ben definite dal protocollo.

Rimossa : l'entità è venuta a conoscenza dell'informazione ma non la diffonde.

La comunicazione avviene in cicli dove i nodi diffondono l'informazione contattandosi a coppie. All'inizio di ogni ciclo un nodo **u** sceglie casualmente fra tutta la popolazione un nodo **v** con il quale può eseguire uno dei seguenti approcci:

Push : **u** diffonde l'informazione a **v**, infettandolo. All'inizio della diffusione, i nodi infetti sono pochi quindi, per un nodo infetto, è alta la probabilità di trovare altri nodi da infettare. Questa probabilità diminuisce con l'aumentare del numero di cicli.

Pull : **u** chiede a **v** se possiede l'informazione aggiornata, in tal caso **u** viene infettato da **v**. Contrariamente all'approccio precedente, la probabilità di essere infettati, che all'inizio è molto bassa data la scarsità di informazioni, cresce con l'aumentare del numero di cicli.

PushPull : se una delle due entità è infetta, entrambe diventano infette. Utilizzando questo approccio l'informazione si diffonde molto velocemente in quanto si sfrutta, prima la velocità dell'approccio push, poi quella dell'approccio pull.

Questi algoritmi sono estremamente **semplici**, **robusti** ed **efficienti** nella distribuzione di un'informazione.

Rimane da definire come un nodo riesca a scegliere casualmente un altro nodo tra tutta popolazione. Dato che la popolazione può essere anche molto numerosa e varia continuamente, è praticamente impossibile mantenere su un nodo la lista di tutti gli altri nodi presenti nella rete in modo da poter sceglierne solo uno. Si implementa quindi un servizio chiamato *peer sampling* che ci permette di scegliere casualmente un nodo della rete mantenendo in memoria una lista limitata.

Infatti, ogni nodo possiede una propria vista locale della rete, cioè un sottoinsieme dei nodi disponibili con i quali si scambia frequentemente delle informazioni tra cui le viste locali. Come si può vedere nella figura 5.1, la vista locale presente e quella ricevuta vengono fuse eliminando i nodi duplicati e selezionando solamente un sottoinsieme della vista risultante che diverrà la nuova vista locale. La scelta dei nodi da mantenere può essere effettuata secondo tre politiche.

Head : sceglie i nodi analizzati più recentemente. Questo approccio permette di eliminare più facilmente riferimenti a nodi non più presenti nella rete ma offre poche possibilità ai nuovi nodi.

Rand : sceglie i nodi in maniera del tutto casuale.

Tail : sceglie i nodi analizzati meno recentemente. Con questo approccio si facilita l'entrata di nuovi nodi ma non si eliminano i nodi non più presenti in rete.

Queste politiche si possono combinare aggiornando una parte dei nodi utilizzando l'approccio head, altri con l'approccio rand e altri con l'approccio tail.

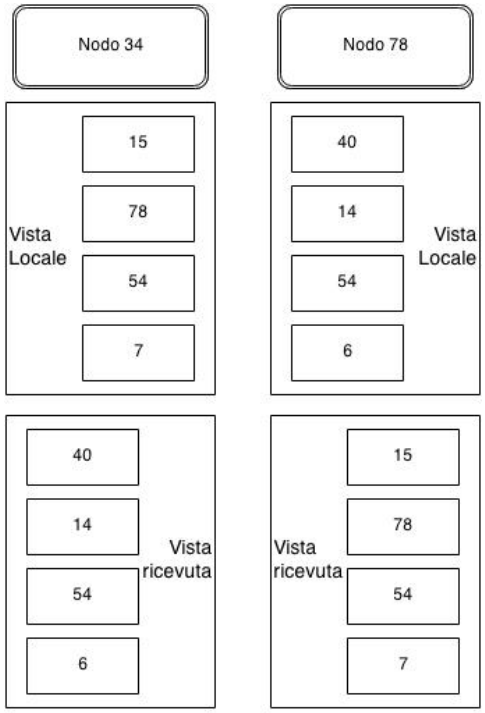


Figura 5.1: Fase 1: scambio delle viste locali

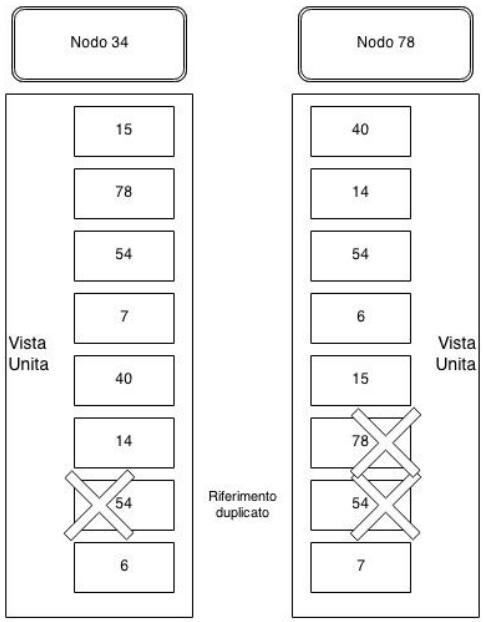


Figura 5.2: Fase 2: fusione delle viste locali con quelle ricevute eliminando i duplicati

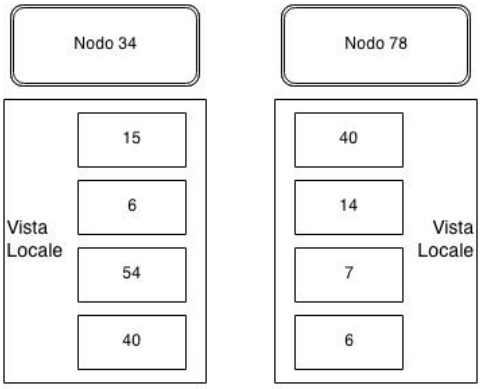


Figura 5.3: Fase 3: scelta di un sottoinsieme della vista unita che definirà la nuova vista locale

5.2 La struttura di un nodo

Un nodo della DHT Izzie, mantiene in memoria un insieme di tabelle, alcune che eredita dalla rete Pastry a cui si ispira(Tabella di routing e LeafSet), altre completamente nuove (BlackList e NodeInfo). Descriviamo dettaglio le componenti che fanno parte di un nodo e che vengono utilizzate per il corretto funzionamento della DHT.

routing table

La routing table è molto simile a quella di Pastry anche se è stata progettata in tre dimensioni. La terza dimensione serve per mantenere una certa ridondanza di indirizzi con lo stesso prefisso.

Nella figura 5.4 è mostrato un esempio di della tabella di routing tridimensionale del nodo 3021. Come abbiamo visto parlando delle vulnerabilità dei sistemi P2P, è molto facile per un attaccante portare a termine un avvelenamento della tabella di routing. Anche se un nodo è sotto attacco, grazie alla ridondanza dei riferimenti, è molto più probabile trovare dei nodi onesti con i quali proseguire la collaborazione per mantenere attivo ed efficiente il servizio offerto dalla DHT.

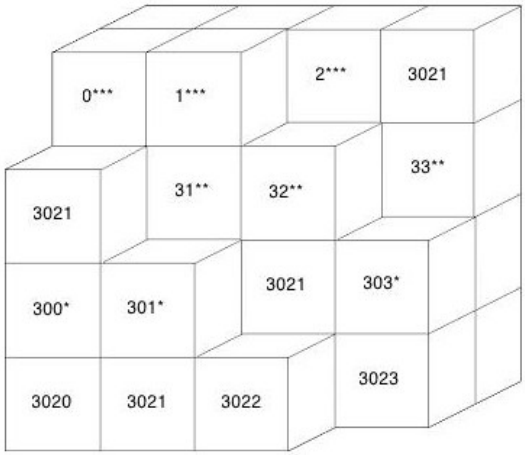
LeafSet

Il LeafSet è lo stesso ereditato da Pastry. Sulla tabella vengono mantenuti i riferimenti ai nodi con ID numericamente più vicino all'id del nodo in considerazione.

BlackList

La BlackList è una lista di nodi che, vengono considerati corrotti dal nodo in questione. Ad ogni peer di cui si viene a conoscenza, si assegna un valore di affidabilità che può variare nel tempo in base al successo o fallimento dei protocolli che vengono eseguiti. Nel caso di fallimenti ripetuti, il valore di affidabilità scende sotto una certa soglia, si suppone che il peer è corrotto e lo si sposta nella BlackList.

Figura 5.4: Routing table tridimensionale



NodeInfo

NodeInfo è una tabella nella quale vengono memorizzati tutte le informazioni di cui si viene a conoscenza sugli altri nodi. Per ogni nodo si mantengono in memoria le seguenti informazioni.

NodeID assegnato al nodo dal controllo Izzie durante la fase di entrata nella rete

Certificato anch'esso generato dal controllo Izzie subito dopo la fase di autenticazione. Mantenendo i certificati degli altri nodi, quando arriva un messaggio firmato, si evita di rivolgersi sempre al controllo per verificare l'autenticità della firma.

Valore di affidabilità che dipende dalla storia delle esecuzioni di protocolli in collaborazione con il nodo.

Statistiche consiste nell'elenco dei protocolli eseguiti insieme al nodo elencando, per ogni protocollo, il numero di successi o fallimenti. Tra le statistiche è anche salvata una lista dei *warnings*. Cioè degli avvertimenti che arrivano da nodi vicini quando questi rilevano un nodo come malevolo.

A differenza del protocollo originale, si può notare che non è presente l'insieme dei nodi geograficamente più vicini (tabella indicata con **N**). Questo è dovuto

al fatto che è stata progettata una nuova tecnica di distribuzione degli ID dei nodi che permette a due nodi geograficamente vicini di avere un ID vicino. Grazie a questa nuova tecnica si elimina un meccanismo macchinoso per il mantenimento del **N**eighborhood set.

5.3 L'assegnazione degli ID nei nodi

In Pastry gli ID dei nodi sono calcolati come hash di indirizzo ip e porta. Questa tecnica comporta una distribuzione pseudo-casuale e uniforme dei nodi sullo spazio delle chiavi. Ciò significa che nodi geograficamente vicini possono avere due ID completamente diversi mentre due nodi situati agli antipodi del mondo possono avere due ID molto vicini. Nonostante le copie ridondanti di un contenuto vengano memorizzati nei nodi con ID molto vicini, non è detto che questi siano anche vicini geograficamente. Questo implica che le copie ridondanti dei file sono equamente distribuite sull'overlay e questo rende la rete maggiormente resistente contro attacchi di tipo Sybil ed Eclipse ma comporta grosse latenze durante il routing. Ipotizziamo che un utente di Roma stia ricercando un contenuto memorizzato in un nodo situato a Milano. Ipotizzando inoltre che l'overlay sia distribuita su tutto il globo, Roma e Milano sono relativamente vicini. Con una distribuzione dei nodi casuale, è possibile che durante il routing, vengano inviati messaggi a New York City, Sydney, Pechino, Il Cairo, Rio De Janeiro, Londra, Tokio, Parigi per arrivare finalmente a Milano. Questo esempio, anche se estremo, mostra come sia possibile che un pacchetto faccia anche più volte il giro del mondo prima di arrivare a destinazione. Si pensi inoltre come nell'esempio riportato siano stati fatti solamente 10 hop ma in realtà, sono molti di più. Infatti utilizzando ID di 128 bit e parametro b uguale a 2, si può arrivare anche a 64 hop accumulando latenze non indifferenti.

É facile rendersi conto che questo problema si può semplicemente ovviare facendo in modo che due nodi geograficamente vicini abbiano due ID vicini. Supponiamo di conoscere le coordinate geografiche del peer, è possibile creare

Tabella 5.1: Dislocazione geografica degli ID dei nodi basata sulle coordinate geografiche

coordinate in binario	0000	0001	0010	0011	0100	0101	0110	0111
0000	0000	0001	0010	0011	0100	0101	0110	0111
0001	0002	0003	0012	0013	0102	0103	0112	0113
0010	0020	0021	0030	0031	0120	0121	0130	0131
0011	0022	0023	0032	0033	0122	0123	0132	0133
0100	0200	0201	0210	0211	0300	0301	0310	0311
0101	0202	0203	0212	0213	0302	0303	0312	0313
0110	0220	0221	0230	0231	0320	0321	0330	0331
0111	0222	0223	0232	0233	0322	0323	0332	0333

l'ID del nodo in funzione di latitudine e longitudine. Rappresentiamo ora latitudine e longitudine come due interi a 32 bit. Indichiamo con x un vettore binario che codifica la longitudine e y un vettore che codifica la latitudine e calcoliamo l'ID del nodo nel seguente modo:

$$ID_{geo} = x_0, y_o, x_1, y_1, ..., x_{31}, y_{31} \tag{5.1}$$

$$ID_{rnd} = SHA - 1(IP + PORT) mod 2^{64} \tag{5.2}$$

$$ID = ID_{geo} + ID_{rnd} \tag{5.3}$$

La tabella 5.1 mostra come sono dislocati gli ID dei nodi utilizzando la tecnica appena descritta. Nell'esempio, si ipotizzano coordinate espresse come sequenze binarie lunghe 4 bit che generano ID lunghi 8 bit mostrati come sequenze in base 4. Grazie a questa disposizione degli ID dei nodi, durante il routing, ogni hop che corrisponde all'aumento della lunghezza del prefisso comune tra la chiave cercata e il nodo corrente, comporta un dimezzamento della distanza geografica tra il nodo che inoltra il messaggio e quello che memorizza la chiave. Immaginando un routing sull'esempio riportato, si accede prima ad un nodo interno al quadrato indicato dalla doppia linea, poi a quello indicato dalla linea singola per accedere infine al nodo cercato.

Questo è vero per i primi $64/2^b$ hop dopodiché il routing torna a seguire degli hop casuali ma con distanze molto brevi. Per avere un'idea di quanto brevi possano essere queste distanze, si consideri che la circonferenza della terrra è lunga circa 40 000 Km e che, esprimendo latitudine e longitudine con 32 bit, possiamo dividere la circonferenza della terra per circa 4 miliardi. Ciò significa che si può raggiungere un margine d'errore di circa 1 cm.

Anche se fosse possibile geolocalizzare ogni peer con una così elevata precisione, ci rendiamo conto che la rete che ne verrebbe fuori avrà dei grossi problemi di distribuzione delle risorse. Dato che le chiavi dei contenuti sono equamente distribuite su tutto lo spazio di indirizzamento mentre gli ID dei file sono concentrati in corrispondenza di coordinate con elevata densità di popolazione, si ottiene un forte sbilanciamento del carico di lavoro tra i peer. Ai peer situati in zone rurali e quelli situati nelle zone costiere, verrebbero assegnati una quantità enorme di chiavi.

Bisogna quindi trovare un compromesso tra l'approccio completamente geolocalizzato come quello appena presentato e quello completamente casuale presente in Pastry. L'idea di base è quella di alternare nell'ID di un nodo sequenze casuali a sequenze derivanti dalla localizzazione. Dividiamo la sequenza di bit calcolata con la localizzazione in 4 sottosequenze di 16 bit ciascuna e indichiamo con $ID^{(i)}_{geo}$ la sottosequenza che inizia dal bit di indice $i \times 16$ Indichiamo invece con ID^{L}_{rnd} una sequenza casuale di bit lunga L. Il nuovo ID del nodo sarà quindi composto come segue:

$$ID = ID^{8}_{rnd} + ID^{(0)}_{geo} + ID^{8}_{rnd} + ID^{(1)}_{geo} + ID^{16}_{rnd} + ID^{(2)}_{geo} + ID^{32}_{rnd} + ID^{(3)}_{geo} \quad (5.4)$$

La prima sequenza casuale è necessaria per ricoprire tutto lo spazio di indirizzamento. Questo comporta una sequenza iniziale di $8/2^b$ hop casuali seguita però da una sequenza di $16/2^b$ hop localizzati. Le successive sequenze casuali permettono una migliore distribuzione delle risorse senza impattare sulle latenze in quanto gli hop avvengono tra nodi sempre più vicini geograficamente.

A questo punto resta solo da analizzare la tecnica per la geo-localizzazione. Fino ad ora si è parlato di localizzazione e coordinate geografiche. In realtà

per i nostri scopi non è importante considerare come vicini due nodi che siano geograficamente vicini ma due nodi che possano comunicare con basse latenze. Per questo motivo, come vedremo successivamente, la localizzazione viene calcolata in funzione delle latenze di trasmissione tra un nuovo peer a quelli già facenti parte della rete.

5.4 La dislocazione delle risorse

La collocazione risorse è un problema da affrontare con cura per garantire una buona efficienza nel mantenere le repliche e una una dislocazione uniforme delle stesse. Considerando l'architettura di un nodo Pastry, e quindi anche di un nodo Izzie, la presenza del LeafSet ci aiuta nel mantenere le repliche nei nodi con ID numericamente vicino. Data la distribuzione degli ID dei nodi, mantenere le repliche solo su un'area delimitata, da una parte velocizza la sincronizzazione delle repliche ma rende tutta la rete più vulnerabile in caso di attacco Eclipse.

Anche in questo caso si cerca di trovare un compromesso che permetta di raggiungere dei vantaggi sia in termini di efficienza che in termini di sicurezza. Il compromesso consiste in una soluzione ibrida nella quale il contenuto viene replicato in un primo momento sullo spazio delle chiavi e in un secondo momento sui nodi geograficamente vicini. Per ogni contenuto esiste una chiave principale K e n chiavi secondarie. Il nodo numericamente più vicino alla chiave principale è l'unico responsabile degli aggiornamenti del contenuto mentre tutti i nodi ai quali è assegnata una chiave sono responsabili del mantenimento delle repliche nei nodi vicini.

Le chiavi secondarie si calcolano facilmente: è sufficiente calcolare una hash di una sottosequenza di bit della chiave principale come mostrato di seguito.

Sia $HASH$ una funzione che mappa una sequenza di bit nello spazio delle chiavi della DHT, $K[j]$ il j-esimo bit della chiave K e $K_{sec}^{(i)}$ dell'i-esima chiave

secondaria del file.

$$K_{sec}^{(i)} = HASH(\ \underbrace{\{k[j]|j \bmod n = i\}}_{\text{sottosequenza della chiave K}}\ ) \tag{5.5}$$

Dopo ogni aggiornamento, il nodo responsabile della chiave principale invierà la nuova versione a tutte le repliche e tutti i responsabili delle chiavi secondarie che si preoccuperanno di mantenere le loro repliche.

Grazie a questa soluzione, è possibile per un peer alla ricerca di un contenuto, conoscere tutte le chiavi del contenuto stesso ed accedere direttamente a quella più vicina.

5.5 La gestione del valore di affidabilità

Come in tutti i sistemi distribuiti, ogni peer ha una propria visione locale delle rete, delle conoscenze su altri nodi e una propria storia formata da interazioni con gli altri peer della rete. Il protocollo presentato prevede che ogni nodo si "crei una propria opinione" sulla bontà e l'affidabilità dei nodi con i quali collabora.

L'idea di fondo è molto simile al comportamento di un essere umano che ha una visione locale del mondo, conosce un sottoinsieme della popolazione e si ricorda i rapporti avuti con gli altri. Grazie a questi ricordi, ogni persona riesce ad crearsi un'opinione del tutto soggettiva su ogni altra persona di sua conoscenza. Questa opinione non è altro che il risultato dei favori e consigli ricevuti ma anche di incomprensioni o sgarbi subiti. Un'altra componente molto importante per giudicare una persona sono le voci e i pettegolezzi circolano tra la gente. Una volta che ci si è fatti un'idea su una persona, ci si comporta di conseguenza. Se qualcuno si comporta continuamente male, si arriverà al punto di perdere la pazienza e tagliare completamente i rapporti. D'altro canto, se qualcuno si è sempre comportato correttamente, si cerca di metterlo in buona luce nei confronti di nuove persone e, nel caso cominci a comportarsi male, gli si offre qualche chance in più prima di chiudere i rapporti.

Ogni nodo della rete Izzie si comporta proprio come una persona semplificando con un solo numero (**valore di affidabilità**) l'opinione maturata nei confronti degli altri peer. Il valore affidabilità è compreso in un determinato range e può cambiare a seconda che un'operazione vada a buon fine o no. Ad ogni operazione è assegnato un valore di importanza che può essere sommato o sottratto dipendentemente dall'esito finale. Se il valore di affidabilità scende sotto una certa soglia, si sposta il nodo dalla routing table alla BlackList.

Proprio come gli umani, anche i peer pubblicizzano le proprie storie negative pubblicando le proprie BlackList utilizzando gli algorimi di Gossip. Quando un peer riceve la BlackList di un suo vicino, non inserisce automaticamente i nodi segnalati nella propria BlackList ma semplicemente decrementa il valore di affidabilità di questi nodi. Questo perché, altrimenti, sarebbe possibile per un attaccante escludere nodi onesti dal servizio semplicemente pubblicando il loro riferimento in una BlackList. Da notare che si riduce il valore di affidabilità di un nodo x solo la prima volta che compare nella BackList di un nodo y. Grazie a questo accorgimento si evita che ricezioni continue della stessa BlackList facciano ridurre troppo il valore di affidabilità dei nodi elencati fino a farli bannare.

Se un nodo è malevolo, è molto probabile che venga escluso da più peer, quindi, ricevendo da diversi vicini un "avvertimento" di *nodo pericoloso*, il valore di affidabilità si riduce fino a comportare l'inserimento del nodo nella BlackList.

5.6 Il routing

Il routing della DHT Izzie nasce come estensione del routing di Pastry. Quando un nodo riceve una richiesta di routing si comporta in modo tale che la richiesta venga inoltrata al nodo, tra quelli conosciuti, il cui ID ha un prefisso in comune con la chiave cercata maggiore dell'ID del nodo stesso.

A differenza di Pastry, la nuova DHT vuole difendersi da potenziali malintenzionati che neghino l'accesso a risorse esistenti. L'architettura della DHT

Izzie ci viene incontro fornendoci tabelle di routing ridondanti contenenti link a nodi ai quali è stato assegnato un giudizio e risorse dislocate sull'overlay con repliche nei nodi vicini. Il routing, dal canto suo, si evolve in una combinazione tra multiple-path[3] e wide-path[22]. Ricordiamo che il multiple-path invia più richieste indipendenti a repliche diverse mentre il wide-path è una forma di routing iterativo molto efficace se le copie ridondanti si trovano su nodi vicini. Nel nostro caso si calcolano le chiavi secondarie del contenuto e si inviano più richieste a k di queste repliche. Ogni richiesta è implementata con un routing iterativo di tipo wide-path. Il nodo che avvia la procedura, Initiator (**I**), è quello che tiene sotto controllo tutta la procedura iterativa. Si parte da un accesso sulla tabella di routing cercando i k nodi numericamente più vicini alla chiave cercata e con un valore di affidabilità non troppo basso e gli si invia una richiesta di routing. Ricevuto il messaggio, i nodi cercano sulle loro tabelle k riferimenti ad altri nodi ancora più vicini alla chiave cercata e inviano questi riferimenti con relativo valore di affidabilità a **I**. L'Initiator riceverà k riferimenti per ognuno dei k nodi contattati. Dei k^2 link ricevuti, si tolgono i duplicati e si scelgono i k più affidabili per iniziare una nuova fase di routing fino a raggiungere i nodi contenenti la chiave cercata. Se un nodo non risponde entro un certo timeout, si decrementa il valore di affidabilità del nodo stesso spostandolo, se necessario, nella BlackList.

L'utilizzo del multiple-path routing per accedere alle diverse repliche dislocate sull'overlay è necessario per difenderci da potenziali attacchi Eclipse. Se un attaccante riesce ad eclissare un'area, tutti le chiavi contenute in essa diventerebbero irraggiungibili. Grazie al multiple-path si accede ad un'altra replica del contenuto riuscendo anche ad individuare quale parte dello spazio delle chiavi è stata eclissata.

Proprio per riuscire ad individuare abbastanza precisamente quale parte dello spazio delle chiavi è stata eclissata, è necessario ricercare un contenuto con il routing ridondante. É stato scelto un routing iterativo perché, altrimenti, con un routing ricorsivo o tracciato non si sarebbe riusciti ad implementare la ridondanza evitando che il numero di richieste crescesse esponenzialmente.

Ipotizziamo infatti un routing ricorsivo dove ogni nodo inoltra una richiesta di routing a 3 vicini. Al secondo hop sulla rete viaggiano già 9 richieste, al terzo 27 e così via fino a congestionare completamente una buona parte dell'overlay. Il wide-path perché riesce ad implementare un routing ridondante mantenendo un numero relativamente basso di messaggi. Ipotizzando un grado di ridondanza di 3, ad ogni passo si inviano 3 messaggi di richiesta e 3 di risposta.

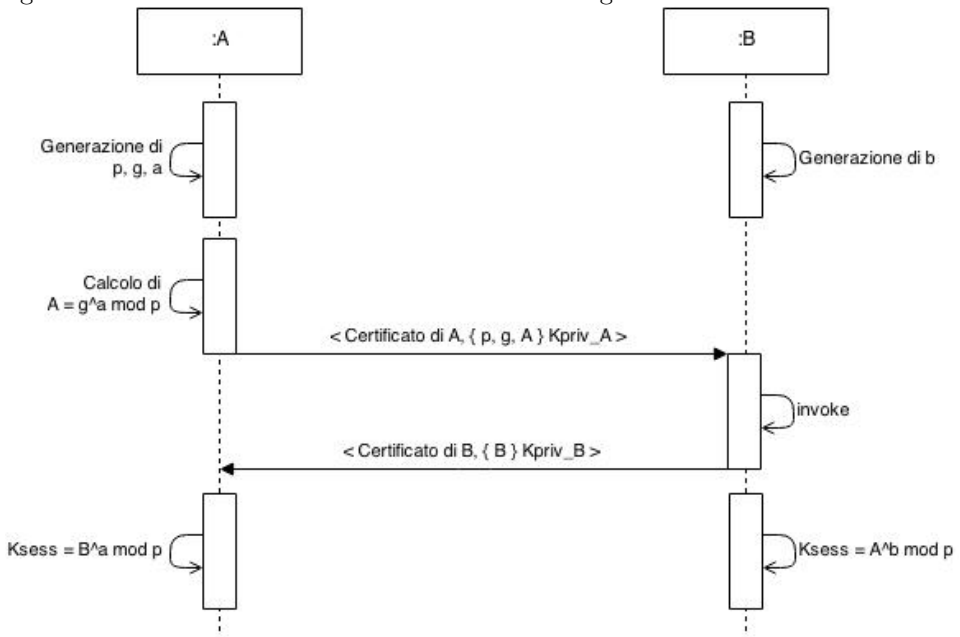
5.7 L'aggiornamento della routing table

Data la semplicità con la quale è possibile subire un avvelenamento della tabella di routing, si ha la necessità di aggiornarla continuamente per evitare che un potenziale malintenzionato riesca ad avvelenarla in maniera tale che, nonostante il routing ricorsivo, non si riesca a raggiungere il contenuto desiderato. Per sfuggire con maggiore efficacia agli effetti di un attaccante, è sempre buona norma utilizzare quanto più possibile delle componenti di casualità nei nostri algoritmi. Per questo motivo, periodicamente, la tabella di routing si aggiorna scegliendo a caso un nodo tra quelli conosciuti con il quale si procede con lo stesso approccio descritto in 5.1 per implementare il servizio di peer sampling. Con il nodo scelto ci si scambia un sottoinsieme casuale dei certificati dei nodi conosciuti con il relativo valore di affidabilità.

Anche questa operazione, se non va a buon fine, implica la riduzione del valore di affidabilità del nodo con cui si comunica. La riduzione è proporzionale alla gravità dell'errore. Per esempio se cade la connessione durante lo scambio è successo un semplice errore di rete e si applica una penalizzazione molto lieve. Se invece avviene un errore nel controllo della firma del certificato, significa che questo è stato alterato e il valore di affidabilità verrà ridotto in modo considerevole.

Tutti i messaggi scambiati in questa fase, così come i messaggi scambiati per la ricerca di una chiave, sono completamente cifrati con una chiave di sessione generata dopo una prima fase di handshake tra la coppia di peer.

Figura 5.5: Protocollo di Diffie-Hellman resistente agli attacchi Man In The Middle



5.8 Le comunicazioni end-to-end

Prima di un qualsiasi scambio di informazioni tra peer, è necessaria una fase nella quale due nodi si scambiano i certificati verificando l'identità dell'altro. Normalmente la verifica dell'identità avviene attraverso un handshake a tre vie nel quale il peer che vuole verificare l'identità altrui genera un nonce, l'altro lo firma e lo rispedisce al mittente. Decifrando la firma con la chiave pubblica dell'utente descritta nel certificato ricevuto e verificando che il nonce ricevuto sia lo stesso generato, si ha la certezza di comunicare con l'utente proprietario del certificato.

La soluzione proposta e schematizzata nella figura 5.5, consiste nell'implementazione del protocollo di Diffie-Hellman nel quale le chiavi pubbliche generate per la sessione sono firmate utilizzando le chiavi RSA private dell'utente. Al termine del protocollo, entrambi i nodi possono generare la stessa chiave di sessione che verrà utilizzata per cifrare la comunicazione.

Questa soluzione è resistente agli attacchi man-in-the-middle e, dato che i messaggi vengono generati dallo stesso nodo che li deve firmare, si evitano

attacchi come il *chosen chipertext attack*. Se un attaccante E volesse utilizzare un certificato di una terza parte T durante la comunicazione con A, non essendo a conoscenza della chiave RSA privata di T, deve firmare la propria chiave pubblica di Diffie-Hellman con un'altra chiave. In questo caso, A, verificando la firma della chiave pubblica ricevuta con la chiave ricevuta nel certificato di T, potrà subito rilevare un furto d'identità bloccando immediatamente la comunicazione.

5.9 L'entrata di un nuovo nodo

Progettando una DHT bisogna sempre tenere di conto il problema del churn rate. Ovvero è sempre possibile che un nuovo nodo entri a far parte della rete o che esca improvvisamente dalla stessa. Come in Pastry non è necessario prendere specifici accorgimenti sull'uscita di un nodo dalla rete. L'unica cosa da considerare è l'uscita di un nodo che è il responsabile delle repliche di un contenuto. In questo caso, attraverso i ping periodici che falliscono, il nodo più vicino alla chiave si accorge dell'uscita del vecchio responsabile e si assume la responsabilità della gestione delle repliche.

L'entrata di un nodo è una cosa molto più complicata in quanto il nuovo nodo non possiede ancora un ID. Ricordiamo che l'ID non è più calcolato come semplice hash di IP e porta ma è una composizione di sequenze casuali e risultato della geo-localizzazione basata sulle latenze dei ping con gli altri peer.

Inoltre, per evitare attacchi Sybil, bisogna garantire che gli ID non possano essere generati dal nodo. La parte dell'ID generata casualmente si può risolvere facendo generare la stringa dal gestore della rete P2P prima dell'emissione del certificato. Il problema sta nel fatto che un malintenzionato potrebbe generarsi latenze ad-hoc per giustificare un determinato ID. L'unico modo che si ha per evitare ciò è fare in modo che le latenze vengano calcolate e firmate da altri peer, il nodo entrante utilizza le latenze calcolate per geolocalizzarsi e generare una parte del proprio ID. Una volta terminata la procedura, si ha

la necessità di farsi generare un certificato dalla componente controllo Izzie la quale controllerà se la parte casuale corrisponde alla stringa emessa in precedenza e la parte generata dalla localizzazione corrisponde alle latenze firmate dai peer che le hanno calcolate.

Seguendo queste line guida definiamo con maggiore precisione il protocollo. Indichiamo con E il nuovo nodo entrante, con C il controllo del servizio e con P_i altri peer già facenti parte della rete.

- E esegue tutta la procedura di login richiesta per accedere alla rete Izzie e riceve un access token come prova dell'avvenuta autenticazione
- E richiede di accedere al servizio P2P inviando il proprio access token
- C controlla l'autenticità dell'access token e genera una stringa casuale di 64 bit che invia ad E insieme ad una lista di nodi già connessi. La stringa generata diventerà la parte casuale che comporrà l'ID di E.
- E conosce solo una parte del proprio ID tra cui i primi 8 bit. Grazie a questi bit si può iniziare la procedura di login molto simile al routing ma ricercando ogni volta solamente un prefisso.
    1. Si contattano i nodi, tra quelli conosciuti, il cui ID è più vicino al prefisso cercato (come primo passo si contattano i nodi P_i il cui contatti sono stati ricevuti da C e si inizia la procedura di login).
    2. Ogni nodo P_i avvia un timer e risponde inviando la propria routing table.
    3. E risponde ai messaggi dei vari P_i con un ACK. Nello stesso momento sfrutta i dati contenuti nella tabella ricevuta per inizializzare le proprie tabelle.
    4. Un nodo P_i, ricevuto l'ACK, interrompe il timer, calcola la latenza di comunicazione con E, la firma e la invia nuovamente ad E
    5. E si estrapola dal node ID le coordinate dei vari P_i e le salva insieme alle latenze ricevute.

- Eseguiti il numero di hop necessari per la ricerca del prefisso, E sfrutta le locazioni dei nodi e le latenze salvate per generarsi una prima approssimazione delle proprie coordinate. Grazie a queste può generarsi altri 16 bit del proprio ID che gli consentono di proseguire con la procedura di login.

Ogni volta che si contattano dei peer basandoci sulla localizzazione, si trovano sempre peer geograficamente più vicini. Ne consegue che le latenze risultano essere più precise fornendo una geo-localizzazione sempre più accurata.

5.10 I Gruppi tra P2P e azienda

Tutte le novità introdotte dalla DHT presentata in termini di sicurezza saranno sicuramente trascurate da un utilizzatore non esperto. L'unica novità il cui utilizzo può essere apprezzato anche da un utilizzatore un po' curioso è l'introduzione di una struttura gerarchica organizzata a gruppi. Questa organizzazione è stata introdotta per adattare la DHT alle esigenze di aziende che possiedono delle infrastrutture tecnologiche proprietarie e un'organizzazione gerarchica per la gestione del personale. Molte delle grandi aziende il personale è organizzato e gestito a gruppi. Per esempio il personale di un'azienda di software, sarà organizzato a gruppi: progettisti, programmatori, analisti, marketing, ecc. I programmatori, a sua volta saranno organizzati a team in base al progetto da sviluppare. Possedere una DHT organizzata a gruppi permette di memorizzare i documenti appartenenti ad un gruppo di lavoro solo sui peer in possesso dei facenti parte del gruppo stesso.

Per progettare la struttura descritta, è necessario risolvere principalmente i seguenti problemi.

Verificare se due peer appartengono allo stesso gruppo

Il problema consiste nel riuscire a progettare un protocollo che garantisca a due peer che si contattano per la prima volta di garantire ad ognuno di essi che l'altro faccia veramente parte di gruppi ai quali dichiara di appartenere. La soluzione più semplice a cui si potrebbe pensare consiste

nell'inserire nel certificato di un nodo la lista dei gruppi a cui fa parte il nodo stesso. Questa soluzione limita notevolmente la flessibilità di tutta l'architettura Izzie in quanto costringerebbe il controllo Izzie alla conoscenza di tutti i gruppi a cui fa parte ogni utente. Un'altra soluzione indipendente dai certificati consiste nell'introduzione di un token casuale generato dal controllo di ogni gruppo e fornito a tutti i client autorizzati. Il possesso del token indica l'appartenenza del peer al gruppo. Durante i continui ping tra coppie casuali di peer, tra le varie informazioni scambiate, ci si scambia anche la lista dei gruppi ai quali ogni peer fa parte, si calcola l'intersezione tra le due liste e, se questa non è vuota, significa che i peer appartengono a gruppi in comune. Bisogna ora verificare che la veridicità dell'affermazione di appartenenza ai gruppi. Questo implica che entrambi i peer sono in possesso dello stesso token che certifica l'appartenenza al gruppo ma non possono scambiarselo perché altrimenti si correrebbe il rischi di fornire un dato sensibile ad un malintenzionato che dichiara di appartenere al gruppo soltanto per venire a conoscenza del token. Si utilizza quindi il protocollo per l'autenticazione Zero Knowledge discusso in 3.4. Entrambi i peer generano una stringa casuale (*nonce*), la cifrano utilizzando il token come chiave e la inviano all'altro. Una volta ricevuta la stringa cifrata, la si decifra e si invia il *nonce* in chiaro al mittente che può controllare la corrispondenza tra il *nonce* generato e quello ricevuto verificando così il possesso del token e quindi l'appartenenza al gruppo da parte dell'altro peer. Tutto questo scambio di *nonce* è completamente nascosto ad una terza parte perché, a questo livello del protocollo, i due nodi hanno già effettuato l'handshake dove hanno verificato l'identità dell'interlocutore e hanno creato un tunnel sicuro e cifrato con una chiave di sessione.

Ottimizzare la ricerca di una chiave all'interno del gruppo

Deve essere possibile che un contenuto memorizzato all'interno di un gruppo sia ottenuto in tempi molto più brevi rispetto a quelli richiesti per la ricerca di una chiave. La soluzione proposta consiste nello sfruttare il

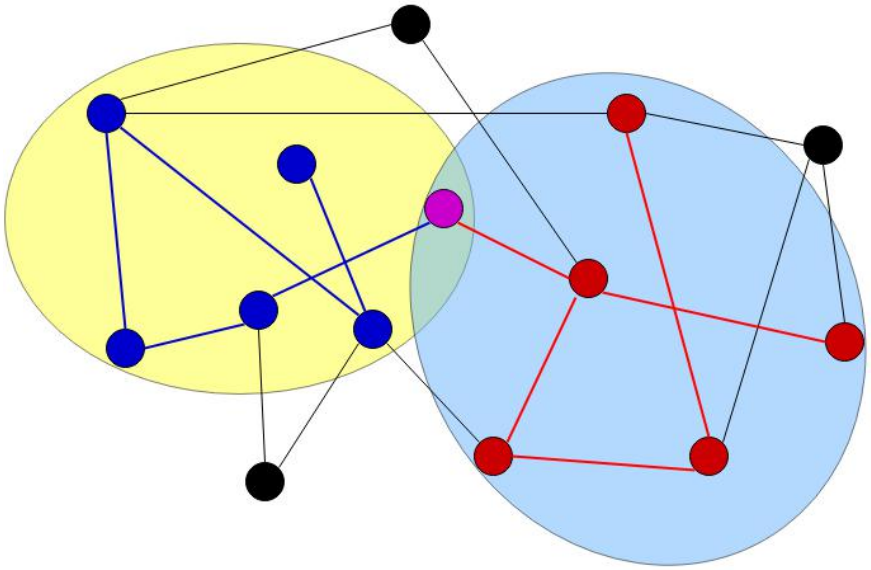
numero limitato dei facenti parte di un gruppo. Grazie alla presenza di un numero relativamente piccolo di nodi è possibile assegnare a loro un nuovo ID più breve di quello assegnato al nodo come parte della DHT di base. Per non confondere i due identificatori, indichiamo con ID l'ID della DHT base mentre con ID_X l'ID assegnato al nodo come parte del gruppo X. In questo modo, un nodo che appartiene a tre gruppi X, Y e Z, verrà identificato con quattro ID: ID, ID_X, ID_Y, ID_Z.

Il routing all'interno di un gruppo verrà effettuato basandoci sullo spazio di indirizzamento piccolo. In questo modo una ricerca di una chiave necessiterà di meno salti portando dei benefici in termini di velocità. Visto che molto probabilmente i nodi dello stesso gruppo sono anche geograficamente vicini, non è necessario che anche gli ID interni al gruppo siano basati sulla localizzazione quindi si possono generare anche casualmente. Il compito della generazione degli ID spetta al gestore del gruppo che dovrà tenere traccia degli ID generati al fine di evitare duplicazioni. Infatti, con ID più brevi, è molto più probabile creare identificatori duplicati nonostante il numero di nodi sia limitato.

Mantenere collegamenti a nodi dello stesso gruppo

Nella tabella di routing riferimenti ai nodi appartenenti allo stesso gruppo potrebbero andare persi a causa del refresh periodico delle entry descritto nei paragrafi precedenti. Bisogna fare in modo che questi non vengano cancellati e inoltre bisogna anche gestire i nuovi ID assegnati ai nodi. Per far questo si ha bisogno di nuove strutture dati per gestire gruppi con i relativi ID. É necessario che per ogni gruppo si mantenga una nuova tabella di routing (non serve sia tridimensionale), un LeafSet e una tabella per mantenere il mapping tra l'ID del nodo come parte della DHT e l'ID del nodo come parte del gruppo. Da notare che le nuove tabelle non occuperanno molta memoria in quanto, sia la tabella di routing e il LeafSet sono di dimensioni ridotte. Quando un peer viene a conoscenza di un'altro dello stesso gruppo, inserirà entrambi gli ID (della DHT e

Figura 5.6: DHT schematizzata con due gruppi.



del gruppo) nelle rispettive tabelle di routing, aggiornerà la tabella per il mapping tra gli ID e inserirà tutte le informazioni sul certificato e valore di affidabilità sulla tabella NodeInfo se non ancora presenti.

Nell'immagine 5.6 è schematizzata una DHT contenente due gruppi. I collegamenti appartenenti ai gruppi sono evidenziati in blu o in rosso. Le linee in nero indicano che due nodi sono collegati solamente sulla DHT base. Come si può vedere possono esistere dei nodi che non appartengono a nessun gruppo mentre, invece, altri nodi possono appartenere anche a più gruppi.

Grazie a questa nuova gerarchia è possibile evitare che i dati non escano dall'insieme dei peer appartenenti al gruppo. Infatti a questi contenuti è associata una chiave breve che sarà mappata nello spazio di indirizzamento relativo al gruppo X e verrà gestita dal nodo il cui ID (ID_X) è numericamente più vicino alla chiave. Di conseguenza, il gruppo è implementato come una vera e propria DHT ridotta che sfrutterà i meccanismi di sicurezza messi a disposizione della DHT base.

Capitolo 6

Izzie: l'implementazione

Il progetto Izzie presentato nei capitoli precedenti è attualmente in via di sviluppo. Nella prima fase di sviluppo è stata tralasciata l'implementazione di un blocco di autenticazione proprietario e si è deciso di integrare client e controllo con i servizi OAuth 2.0 offerti da Facebook e Google. L'implementazione delle funzionalità dei controlli è parte del lavoro di un'altra tesi mentre, in questa tesi ci si è anche dedicati allo sviluppo del client.

Mentre nel capitolo precedente si sono discusse le problematiche relative alla gestione della DHT, in questo capitolo ci concentreremo sugli aspetti relativi all'implementazione della stessa. Data la mole di lavoro si è deciso di eseguire un approccio bottom-up quindi, come prima fase di sviluppo, si sono implementati tutti i protocolli e le funzionalità utili per l'integrazione con i blocchi di autenticazione e controllo e la comunicazione tra peer.

É stata quindi implementata una libreria Java che consenta all'utente di autenticarsi con i due Authentication Provider supportati, interagire con il controllo per ottenere il certificato e comunicare in completa sicurezza con altri client. La scelta di Java come linguaggio per l'implementazione del client è stata dettata dalla portabilità del programma su più sistemi operativi.

La libreria sviluppata richiede che il nodo sia già in possesso del certificato rilasciato dal controllo dopo aver verificato l'identità dell'utente che si è connesso. Durante lo sviluppo ci si è occupati dell'implementazione del protocollo

OAuth e la creazione di un client REST per rendere necessaria l'integrazione con blocco di autenticazione e blocco del controllo.

Si è quindi definita una struttura che permetta la creazione di un tunnel mediante la quale due peer possano comunicare in completa sicurezza. La gestione della comunicazione end-to-end è strutturata a livelli dove, nel livello più basso è stata implementata una componente che si occupa di interagire con le socket TCP e UDP per stabilire connessioni diretta tra peer mentre nel livello più alto si posizionano i moduli per la gestione lato client dei servizi offerti dalla rete Izzie.

In questo capitolo vedremo in dettaglio come il client interagisce con gli altri blocchi e analizzeremo le componenti utili per la comunicazione tra peer e i problemi che si devono risolvere. Uno di questi problemi nasce senza dubbio dalla presenza dei router NAT che, a meno di impostazioni specifiche, bloccano le connessioni entranti in una rete locale. Per questo motivo, in un primo momento presenteremo le tecniche utilizzate per stabilire connessioni tra peer nonostante la presenza dei NAT, poi analizzeremo il formato Json utilizzato nello scambio dei dati prima di poter poi entrare nel dettaglio delle problematiche affrontate durante l'implementazione

Come caso d'uso della libreria implementata è stato sviluppato un servizio di chat aziendale sicura. Come tutte le applicazioni che si possono integrare sull'architettura Izzie, anche la chat aziendale richiede l'introduzione di un controllo. Dato che lo sviluppo delle funzionalità del controllo, almeno in questa prima fase, è un lavoro indipendente da quello presentato in questa tesi, è stato sviluppato uno stub che implementa un sottoinsieme delle funzionalità di un controllo.

6.1 Il superamento dei router NAT

Come è noto, un router NAT è uno strumento in grado di stabilire una comunicazione tra due host situati l'uno all'interno di una rete privata e l'altro all'esterno delle stessa. Un grosso limite dei NAT consiste nel fatto che solo

l'host nattato è in grado di aprire una connessione TCP oppure inviare per primo un pacchetto UDP verso l'altro. Per questo motivo, i NAT costituiscono un ostacolo per lo sviluppo di reti P2P in quanto, per queste ultime, si ha la necessità che entrambi i peer si possano contattare reciprocamente in qualsiasi momento. Per ovviare a questo problema, un protocollo P2P si avvale dell'aiuto di una terza parte, facilmente accessibile dalla rete che indicheremo con il nome di buddy server. Questa componente può essere centralizzata o distribuita infatti, ogni nodo può avere come buddy un'altro peer della rete. Ogni buddy mantiene aperta una connessione con tutti gli host a cui offre il servizio. In base a quanto detto precedentemente questa connessione è aperta dal peer quando entra a far parte della rete e permette al buddy di inviare in qualsiasi momento un messaggio al peer connesso.

Analizziamo i vari scenari che si possono presentare quando un peer vuole iniziare una comunicazione verso un'altro peer.

- Entrambi i peer sono connessi direttamente a internet senza utilizzare nessun meccanismo di NAT
- Un peer è connesso al NAT e l'altro ha l'accesso diretto a internet
- I peer appartengono a due reti locali distinte e nascoste da un NAT
- I peer appartengono alla stessa rete locale.

Il primo e ultimo scenario sono ovviamente i più semplici in quanto, senza nessun accorgimento particolare, entrambi gli host sono facilmente accessibili l'un dall'altro.

Il secondo scenario può cambiare a seconda di quale host vuole iniziare una comunicazione. Infatti il peer situato in una rete privata può aprire facilmente una connessione verso l'altro, ma non vale il contrario. Ipotizziamo di avere due host A e B dove solo B si trova in una rete privata e A è accessibile direttamente da internet. Di conseguenza solo B può aprire una connessione verso A e, nel caso in cui A voglia comunicare con B bisogna attuare il protocollo di *connectionReversal* (figura 6.1). Il protocollo sfrutta il fatto che il nodo B

Figura 6.1: Protocollo per il connection traversal

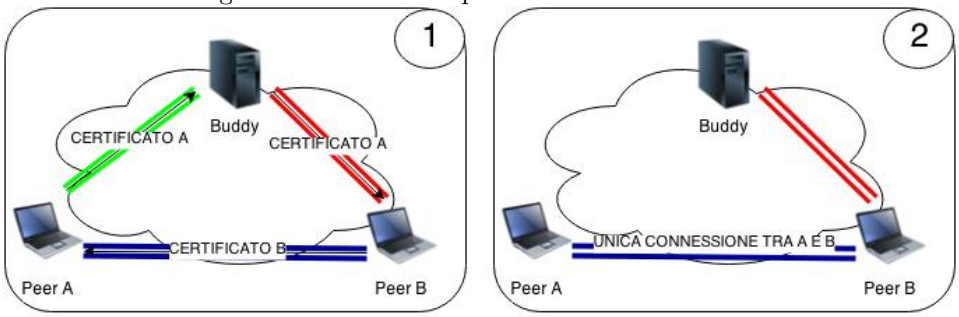
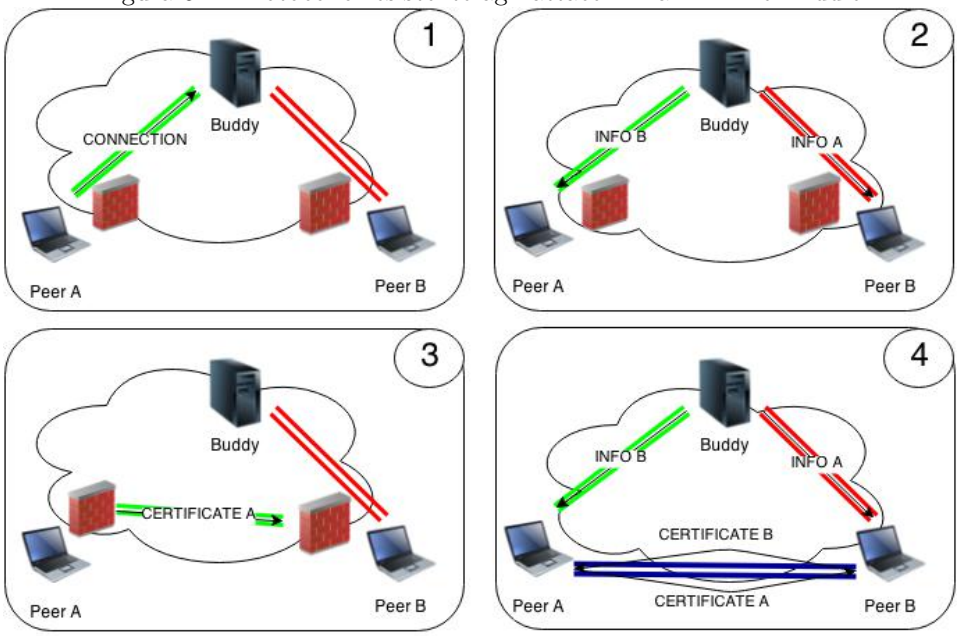


Figura 6.2: Protocollo resistente agli attacchi Man In The Middle



ha una connessione attiva verso il proprio buddy. Esso è costituito da tre messaggi:

1. $A \rightarrow buddy$: B comunica al buddy di B l'intenzione di comunicare con B
2. $buddy \rightarrow A$: il buddy inoltra a B il messaggio di A di stabilire una connessione inviando l'indirizzo IP e porta di A
3. $B \rightarrow A$: B sfrutta le informazioni ricevute dal buddy per avviare la comunicazione con A

Il terzo scenario, infine, è il più complesso in quanto entrambi i nodi sono situati a monte di un router NAT. Anche in questo caso è richiesto l'ausilio del server per implementare il protocollo di *HolePunching*[17]. Supponendo che un nodo A voglia stabilire una comunicazione UDP con B, questa avviene nelle seguenti fasi mostrate in figura 6.2.

1. A comunica al buddy S l'intenzione di comunicare con B.
2. S risponde ad A inviando l'indirizzo pubblico e privato di B e nello stesso momento invia a B l'indirizzo pubblico e privato di A
3. Sia A che B, appena ricevuto il messaggio da S, provano a stabilire una connessione su entrambi gli indirizzi (pubblico e privato) dell'altro peer.

Se entrambi i peer si trovano a monte dello stesso NAT, riceveranno un messaggio dall'altro host direttamente attraverso la rete privata. In caso contrario, una volta che A ha inviato un pacchetto verso l'indirizzo pubblico di B, il router di A aggiunge sulla propria tabella NAT una nuova associazione tra l'indirizzo IP privato di A e quello pubblico di B. In questo modo, quando B invierà un pacchetto verso il router NAT di A, questo non scarterà il messaggio e lo inoltrerà al peer A.

Come si può vedere anche dalle immagini in fig. 6.1 e 6.2, ogni nodo, appena è riuscito a stabilire una connessione verso l'altro peer, invia subito il proprio certificato in modo tale da poter successivamente firmare i messaggi secondo quanto specificato dal protocollo.

Purtroppo, dai risultati riportati in [17], l'hole punching non funziona correttamente su tutti le marche di router in commercio. Per questo motivo, anche se ridotta, esiste sempre la possibilità che la comunicazione tra due peer debba passare attraverso un buddy. Per ridurre ulteriormente questa possibilità, è possibile, se il router lo supporta, rendere un peer accessibile dall'esterno attraverso il protocollo di "Port Mapping". Questo permette ad una applicazione installata su un computer della rete locale di creare un collegamento sul router in modo che tutte le richieste indirizzate ad una determinata porta vengano inoltrate al dispositivo utilizzato.

6.2 L'apertura delle porte su NAT attraverso NAT-PMP

Per rendere i collegamenti tra peer più veloci e sicuri, è necessario evitare, se possibile, di far passare tutti i pacchetti attraverso una terza entità. La tecnologia ci viene incontro in quanto esiste un protocollo che, anche se usato da alcuni anni, è stato formalizzato solo nel 2013 sotto il nome di NAT-PMP[4].

Il protocollo consiste nello scambio di una serie di messaggi SOAP[42] con lo scopo di:

- richiedere le informazioni del gateway di default;
- richiedere se la porta esterna su cui si vuole creare un collegamento è libera;
- richiedere la creazione del collegamento

Il protocollo non è stato implementato direttamente ma è stata utilizzata la libreria "weupnp[33]", una libreria open source rilasciata con licenza GNU. La prima operazione eseguita dal client quando va in esecuzione, è proprio l'apertura delle porte per sapere se e a quale indirizzo si è accessibili dalla rete Internet.

In un primo momento il client, inviando un messaggio in broadcast, esegue una ricerca dei Gateway di rete. Se nella rete locale esiste un Gateway con PortMapping abilitato, questo risponde inviando le proprie informazioni tra cui indirizzo della rete locale e indirizzo pubblico. Il client, a questo punto, genera un numero di porta casuale e verifica l'esistenza di un mapping sul numero di porta generato. Se il numero di porta è libera si aggiunge un nuovo mapping rendendo così il client accessibile dall'esterno sulla porta dichiarata.

6.3 Il formato JSON

Il formato JSON **JavaScript Object Notation**[13] è un formato progettato per lo scambio dati tra client e server. Nasce come formato per la comunicazione asincrona tra JavaScript e il server nella programmazione in AJAX. JSON

è stato proposto come formato per lo scambio dati da Douglas Crockford[8], il quale ne ha anche definito le specifiche. Egli lo definisce così:

> JSON (JavaScript Object Notation) è un formato leggero per lo scambio di dati, facile da leggere e scrivere per gli esseri umani e facile da generare e analizzare da parte delle macchine. La sintassi riprende il modo di dichiarare le strutture dati in Javascript, definito nelle specifiche ECMA-262, 3a edizione, del dicembre 1999. JSON è un formato di testo completamente indipendente dal linguaggio ma che usa convenzioni già familiari ai programmatori dei linguaggi derivati dal C, tra cui C, C++, C#, Java, JavaScript, Perl, Python e molti altri. Queste caratteristiche rendono JSON un linguaggio ideale per lo scambio di dati.

La sintassi di JSON, prende origine dalla sintassi degli oggetti letterali in JavaScript definiti così:

```
var JSON = {
  proprieta1: 'Valore',
  proprieta2: 'Valore',
  ...
  proprietaN: 'Valore'
};
```

Listing 6.1: Oggetto letterale in JavaScript

Si tratta di coppie di proprietà/valori separate dalla virgola ad eccezione dell'ultima. L'intero oggetto viene racchiuso tra parentesi graffe. A differenza della stessa notazione JavaScript, che può contenere anche funzioni e valori complessi, JSON ammette solo valori semplici ed atomici, tra cui:

- Booleani (True e False);
- Interi, Reali, Virgola mobile;
- Stringhe racchiuse da doppi apici;
- Oggetti letterali (sequenze coppie chiave-valore separate da virgole racchiuse tra parentesi graffe);

- Array (sequenze ordinate di valori, separati da virgole e racchiusi tra parentesi quadre [ ]);
- Null

Data la limitazione sui tipi di valori utilizzati, Json non può essere utilizzato per rappresentare dati o meta-dati complessi. Per queste applicazioni è più indicato l'utilizzo del formato XML[40]. Grazie alla sua semplicità, Json, è sempre più utilizzato ed è stato scelto come formato dati nella maggior parte delle comunicazioni dell'architettura Izzie. Infatti, in tutte le API Rest fornite dai vari controlli, è specificato il Json come formato dati richiesto.

6.4 Le componenti

Come si è accennato nella parte introduttiva di questo capitolo, il progetto è in via di sviluppo e il lavoro riguardante la progettazione e implementazione delle funzionalità dei controlli procede indipendentemente dal lavoro presentato in questa tesi. Per rendere funzionante la libreria è stato implementato uno stub che implementa alcune funzionalità dei controlli.

Nella rete attiva in questo momento è possibile individuare le seguenti componenti.

I server OAuth 2.0

Si tratta del fornitore del servizio di autenticazione attraverso il quale l'utente è registrato. Come anticipato precedentemente gli Authentication Provider supportati al momento sono Google e Facebook.

Il buddy server

É un server unico e centralizzato che si comporta come buddy di tutti i peer situati dietro NAT. Si è preferito un punto centrale per poter analizzare la parte di traffico che viene inviata ai buddy al fine di semplificare la fase di debugging.

Il gestore P2P É l'entità che implementa il controllo Izzie. Le funzionalità necessarie sono limitate alla rilevazione degli indirizzi pubblici dei client e la generazione dei certificati.

Il gestore della chat É il controllo del servizio. Si occupa di mantenere l'elenco di tutti i peer connessi con i relativi certificati. Ogni peer che vuole sfruttare il servizio chat, si deve connettere al gestore e inviargli il proprio certificato insieme allo stato dell'utente(online, offline, occupato, ecc) firmato. Ogni variazione dell'elenco degli utenti connessi viene notificata in broadcast a tutti i nodi.

Il client É l'entità che si interfaccia con il resto dei componenti elencati. I compiti del client sono quelli di stabilire, se possibile, una connessione diretta ed efficiente tra due peer qualsiasi e creare un tunnel cifrato con chiavi di sessione one-time.

Come si può dedurre dalla descrizione dei componenti, ogni volta che il client vuole stabilire qualsiasi comunicazione, per prima cosa si presenta inviando il proprio certificato. Vediamo quindi nel prossimo capitolo quali sono le informazioni contenute nel certificato che contraddistingue l'identità di un utente.

6.5 I certificati

Ogni nodo della rete è contraddistinto da un certificato nel quale sono memorizzate tutte le informazioni che lo riguardano. Qualsiasi peer, per stabilire una connessione sicura con un altro peer, ha bisogno di conoscere il suo certificato. Ogni certificato contiene alcune informazioni riguardanti il client e altre riguardanti l'utente che lo utilizza.

NodeID< *string* > : ID casuale e unico del client

PublicAccess< *boolean* > : indica se il client è accessibile o no da Internet. Il valore di questo flag è vero se e solo se il client ha un indirizzo IP pubblico oppure ha attivato il port mapping sul proprio router.

IsBehindNAT< *boolean* > : indica se il client è situato dietro un router NAT. Il valore è calcolato confrontando l'indirizzo pubblico dichiarato dal client con quello riscontrato dal Controllo.

LocalAddress< *string* > : indirizzo IP assegnato al client nella rete locale. Il campo è opzionale e deve essere impostato solo se il valore "IsBehindNAT" è uguale a vero.

LocalTCPPort< *int* > : porta TCP su cui è in ascolto il client sulla propria rete locale. Anche questo campo ha senso solo se il nodo è situato dietro un NAT.

LocalUDPPort< *int* > : porta UDP su cui è in ascolto il client sulla propria rete locale. Anche questo campo ha senso solo se il nodo è situato a monte un NAT.

PublicAddress< *string* > : indirizzo IP pubblico del client o del router NAT se il peer si trova in una rete locale. La presenza di questo campo è utile solo se il peer è accessibile da Internet.

PublicTCPPort< *int* > : porta TCP su cui è in ascolto il client sulla rete pubblica o, se il peer si trova su una rete locale, è a la porta TCP esterna sulla quale è stato definito il "Port Mapping".

PublicUDPPort< *int* > : porta UDP su cui è in ascolto il client sulla rete pubblica o, se il peer si trova su una rete locale, è a la porta UDP esterna sulla quale è stato definito il "Port Mapping".

BuddyServer< *string* > : indirizzo IP e porta TCP del buddy associato al nodo. Il campo è necessario se il nodo non è accessibile da Internet.

UserID< *string* > : ID dell'utente che utilizza il peer.

UserName< *string* > : nome dell'utente.

AuthProvider< *string* > : codice dell'Authentication Provider utilizzato per l'autenticazione.

AuthProvUserID< *string* > : ID che l'Authentication Provider ha assegnato all'utente.

PublicKey< *string* > : chiave pubblica dell'utente.

Signature< *string* > : Firma di tutte le informazioni del certificato effettuata con la chiave privata del gestore P2P.

Come si può vedere, il certificato contiene tutte le informazioni per raggiungere un nodo e stabilire una comunicazione sicura. Per questo motivo, ogni sessione di comunicazione è preceduta da un handshake nel quale i due nodi si presentano scambiandosi il proprio certificato.

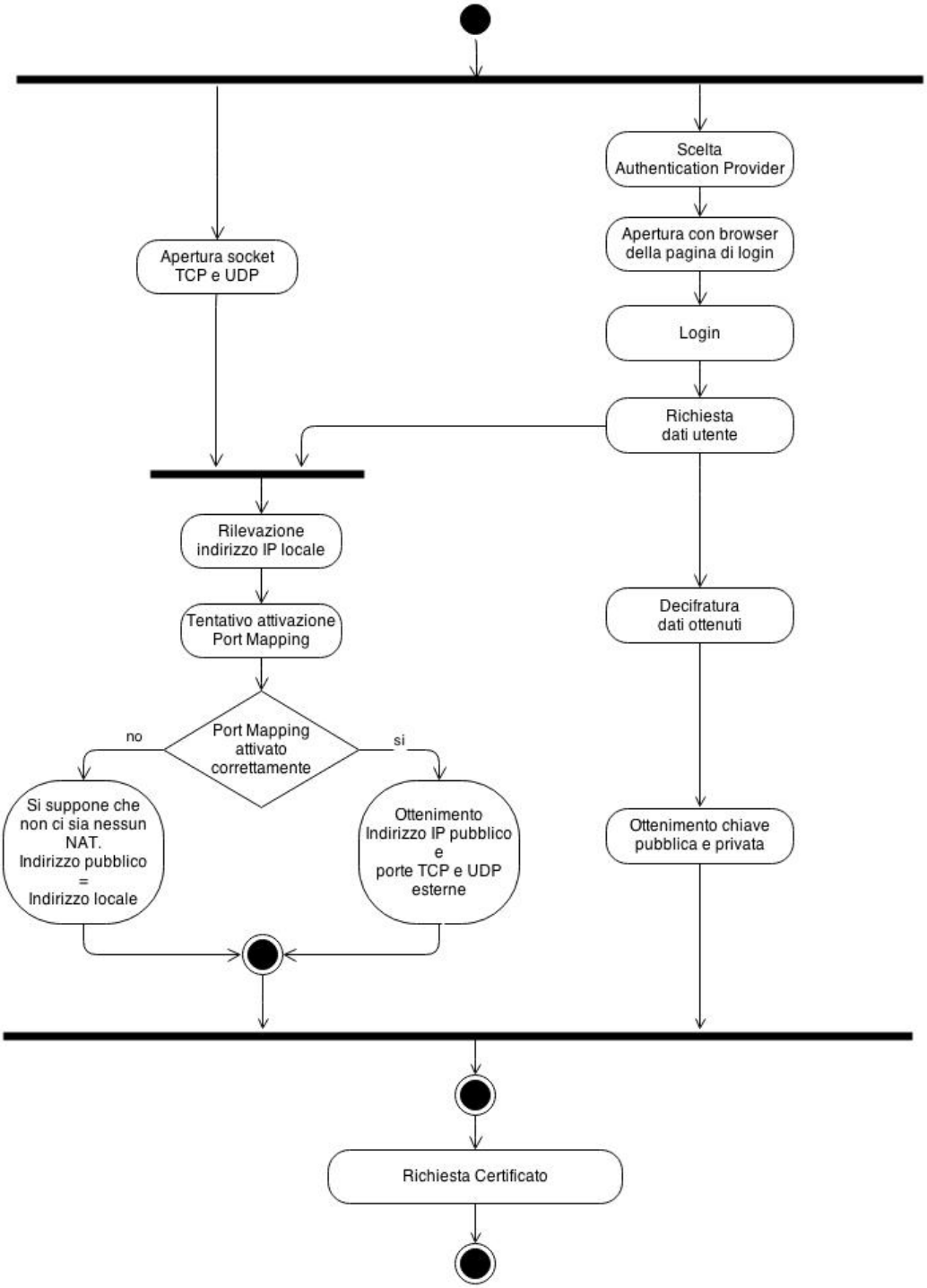
6.6 L'avvio del client

Vediamo ora tutti i passi che vengono eseguiti per ottenere un certificato come quello descritto nella sezione precedente. Si consideri che l'utente abbia già effettuato la registrazione e quindi il gestore P2P (che ricordiamo implementa parte delle funzionalità del Controllo), sia già a conoscenza della chiave pubblica dell'utente e possieda la chiave privata cifrata con la masterKey dell'utente stesso.

La figura 6.3 ci mostra tutti i passi effettuati per ottenere i dati necessari per il rilascio di un certificato. É possibile individuare due procedure, a sinistra è mostrata quella per raccogliere le informazioni sul dispositivo mentre a destra quella per raccogliere le informazioni sull'utente.

Iniziamo descrivendo le operazioni necessarie per ottenere le informazioni sull'utente. Appena si lancia l'applicazione, l'utente deve scegliere quale Authentication Provider utilizzare per eseguire il login. Effettuata la scelta, il client apre un browser che mostra una finestra che indirizza la pagina di login esposta dal servizio scelto. L'utente in questo modo interagisce direttamente con l'Authentication Provider attraverso una connessione SSL senza la possibilità che il client venga in possesso delle credenziali dell'utente.

Figura 6.3: Sequenza di azioni all'avvio del peer



Effettuato il login, il browser verrà reindirizzato su una pagina contenente il token di accesso che viene intercettato dal client. Il client può così inviare una prima richiesta al gestore P2P fornendo il token di autenticazione e ricevendo una stringa cifrata contenente le informazioni che l'utente aveva salvato sul gestore P2P.

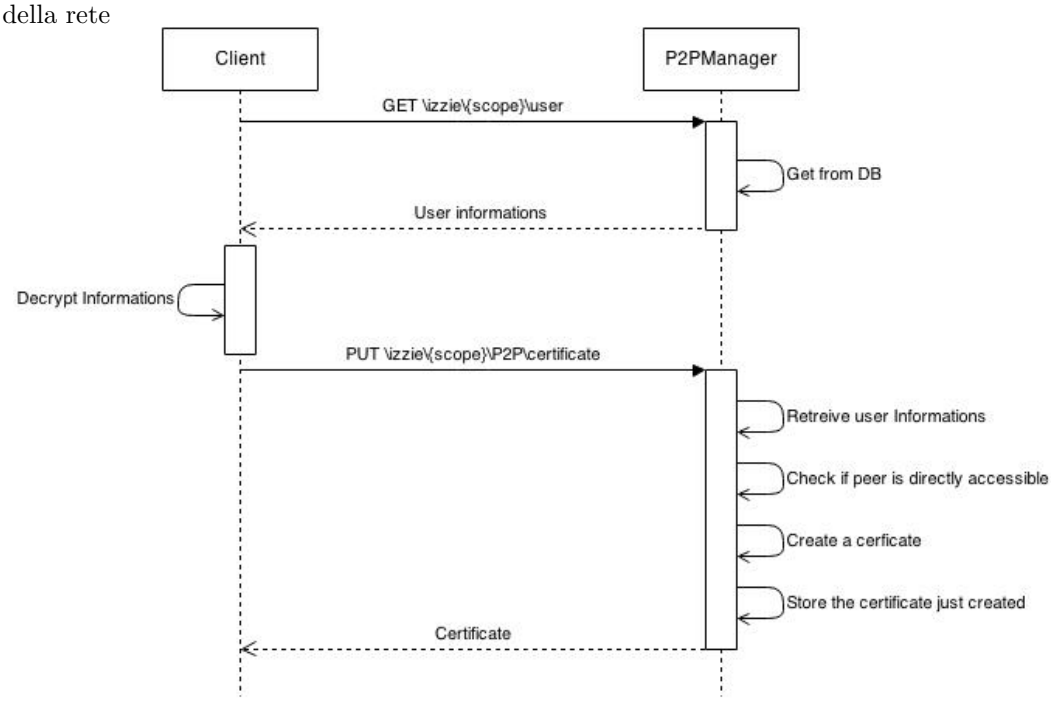
Il client avrà la necessità di decifrare la stringa ricevuta quindi richiede all'utente la propria masterPassword. Sulla password inserita verranno applicate in sequenza una serie di funzioni hash fino a trasformarla in una chiave per la cifratura simmetrica con la quale si possono decifrare le informazioni ricevute tra cui le chiavi RSA generate durante la registrazione.

Durante la richiesta delle informazioni sull'utente, il client accede alle informazioni della socket e ottiene l'indirizzo IP locale della connessione. Questo ultimo passo è servito per evitare di implementare il codice per accedere a tutte le interfacce di rete e ottenere quella di default. Il compito è così demandato al sistema operativo.

La rilevazione del proprio indirizzo IP locale è un passo della procedura che raccoglie le informazioni sul dispositivo. Infatti, come mostra il diagramma sulla figura 6.3, il client, prima di questo passo, apre due socket UDP e TCP e crea un binding su due porte che possono essere casuali o possono essere dichiarate nella configurazione del client. Anche se sulla configurazione è previsto l'utilizzo di una determinata porta, se il binding fallisce perché la porta è già occupata da un altro processo, l'applicazione sceglierà una porta casuale e continuerà la propria esecuzione.

A questo punto, il client prova ad eseguire il protocollo di Port Mapping per assegnare al router il collegamento tra due numeri di porta esterne e le due socket (TCP e UDP) aperte precedentemente. Se il protocollo va a buon fine, si richiede al router l'indirizzo IP pubblico altrimenti si suppone di trovarsi sulla rete pubblica e si ritiene che l'indirizzo IP pubblico sia lo stesso di quello locale. In realtà esiste anche il caso in cui il router non supporti il protocollo di Port Mapping oppure è disabilitato. Per avere una conoscenza precisa sulla situazione della connessione del peer è necessario confrontare l'indirizzo IP

Figura 6.4: La sequenza di chiamate REST effettuate dal client prima di entrare a far parte della rete



pubblico e locale con quello rilevato da un'entità che siamo sicuri non essere a monte di un NAT. Il compito della rilevazione dello stato della connessione del peer viene così delegato al gestore P2P che, ricevendo dal client tutte le informazioni che esso ha raccolto, può generare il certificato.

Nello specifico, il gestore P2P, ha bisogno di ricevere il token di accesso, il codice del provider utilizzato per l'autenticazione, la chiave pubblica dell'utente, l'indirizzo locale del client e quello dichiarato come pubblico. Grazie al token di accesso e il codice del provider utilizzato, il server può ottenere le informazioni sull'utente da inserire sul certificato ovvero l'ID dell'utente, il nome, il provider di autenticazione utilizzato e l'ID che questo provider ha assegnato all'utente.

Rimane ancora da definire come vengono elaborate le informazioni riguardanti il client. Come accennato in precedenza, il NodeID viene calcolato come una sequenza casuale. Se la stringa definita è stata già assegnata ad un altro peer se ne calcola un'altra.

Controllando l'indirizzo IP remoto della socket sulla quale è stata ricevuta la richiesta, è possibile stabilire se il nodo è accessibile e se è situato a monte di un NAT. Confrontando l'indirizzo IP del client rilevato dalla socket con l'indirizzo IP locale è possibile stabilire la presenza di router NAT mentre, confrontandolo con l'indirizzo dichiarato come pubblico, si verifica se il nodo è accessibile da internet. Quest'ultimo caso si verifica quando il nodo non è dietro NAT oppure si è riusciti a configurare un Port Mapping.

Nel caso in cui il nodo non sia accessibile da internet, il gestore P2P gli assegna un buddy. Come si era detto in precedenza, durante la fase di testing il buddy assegnato a tutti i peer sarà un unico server nel quale è installato demone con la funzione di inoltrare i pacchetti al reale destinatario.

Il gestore P2P compila così tutti i campi del certificato, lo firma e lo trasmette al client insieme alla propria chiave pubblica. Il client può così verificare la firma apposta sul proprio certificato e può cominciare la normale attività.

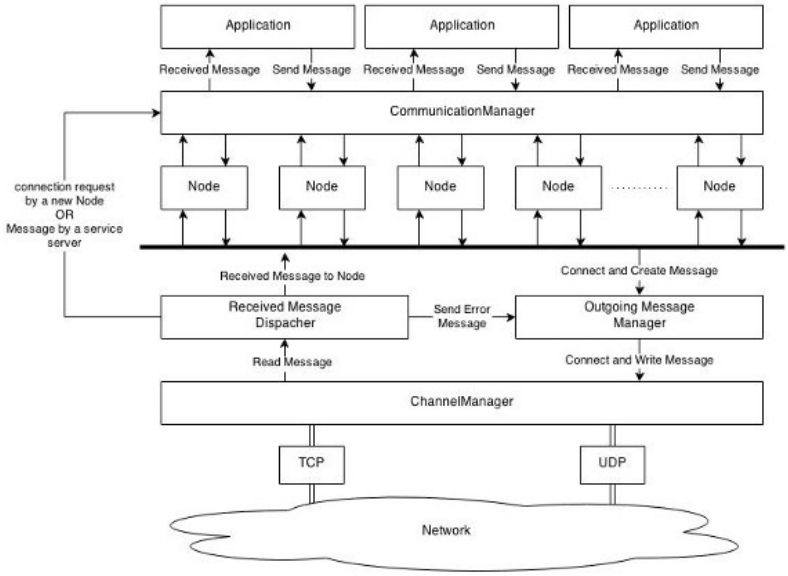
6.7 L'architettura

Il supporto che abbiamo implementato deve consentire ai programmatori di un servizio di comunicare in modo sicuro con un altro utente senza occuparsi di implementare funzionalità legate alla sicurezza.

La libreria sviluppata è strutturata a livelli, dove al livello più basso troviamo l'interazione con le socket UDP e TCP e al livello più alto ci sono le applicazioni. Lo schema in figura 6.5 mostra dove le componenti principali del sistema si localizzano nello stack e come comunicano tra loro.

Il servizio quindi si trova a livello applicativo e interagisce con un solo componente: il **CommunicationManager**. Questo è il componente che si occupa di istanziare e gestire tutti gli altri componenti e inoltrare i messaggi al corretto destinatario: i messaggi da inviare saranno inoltrati al gestore delle interazioni con un determinato nodo (componente **Node**) mentre quelli ricevuti al servizio al quale sono sono destinati.

Figura 6.5: L'architettura implementata per la comunicazione End-to-End sicura



La componente **Node** si occupa di gestire tutta la sequenza di messaggi scambiati con un altro peer della rete. Questo oggetto è il responsabile della creazione e gestione del canale sicuro che collega i due peer. I **Node** interagiscono con il **CommunicationManager** attraverso il quale ricevono i messaggi da inviare e al quale comunicano il messaggio ricevuto.

A basso livello, invece, i **Node** interagiscono con due componenti diversi: il primo, **OutgoingMessageManager**, è il gestore dei messaggi uscenti e si occupa di impacchettare i messaggi da inviare; il secondo, **ReceivedMessageDispatcher**, è il componente che si occupa di analizzare i messaggi entranti e indirizzarli al giusto gestore.

Nel livello più basso dello stack c'è il **ChannelManager**. Questo componente si occupa dell'interazione diretta con i canali UDP e TCP per gestire le trasmissioni, connessioni e timeout.

6.8 Il ChannelManager

Il ChannelManager è l'entità che si trova nel livello più basso dello stack implementato. Esso si occupa, in un unico thread, di gestire tutte le interazioni con le socket: aprire o accettare connessioni TCP e inviare o ricevere messaggi UDP e TCP. La componente è stata implementata come una classe che mette a disposizione alcuni metodi per ricevere comandi dai livelli più alti e crea degli eventi che possono essere gestiti registrando i relativi "Listener".

Tra questi comandi i più importanti che possono essere inviati al ChannelManager riportiamo:

- Connect per stabilire una connessione verso una determinata coppia indirizzo IP e porta. É possibile specificare anche il numero di tentativi in caso non si riesca a stabilire la connessione immediatamente. Come parametro del comando *connect* è possibile inviare il Listener per gestire tutti gli eventi legati alla connessione come **connectionEstablished**, **connectionFailed** e **connectionEnded**. Il primo indica che la connessione è stata stabilita con successo, il secondo indica che sono stati effettuati tutti i tentativi dichiarati ma non si è riusciti comunque a stabilire la connessione e il terzo viene sollevato quando la connessione che era stata stabilita in precedenza viene chiusa.

- **closeConnection** per chiedere al ChannelManager di chiudere una connessione con una socket specificata.

- **sendTCPMessage** e **sendUDPMessage** per inviare un messaggio ad un determinato destinatario rispettivamente tramite la socket TCP o UDP. Insieme al comando si specifica un Listener per gestire l'evento causato dal fallimento del messaggio stesso.

- **subscribeReadEvent** è invece il comando che permette ad una entità esterna di sottoscriversi all'evento di messaggio ricevuto registrando un proprio Listener.

- **setAcceptHandler** è simile al precedente ma è utilizzato per registrarsi all'evento di connessione accettata.

Per gestire tutte le socket in un unico thread rendendo il componente il più efficiente possibile, si è utilizzata la tecnologia introdotta da Java con la versione 1.4: Java NIO[6]. La tecnologia permette la gesione di socket non bloccanti attraverso un unico oggetto Selector. Sulla Selector si registrano le varie socket da gestire dopodiché si chiama il metodo select che blocca l'esecuzione del thread finché non si verifica un evento su una delle socket registrate. Quindi il metodo restituisce un insieme delle socket dove è avvenuto un evento demandando il compito della gestione all'applicazione. I principali eventi da gestire sono quelli elencati qui di seguito.

Accept Quando la socket TCP che sta in ascolto sulla rete riceve una nuova richiesta di connessione si verifica l'evento Accept. Il ChannelManager gestisce l'evento creando una nuova connessione e aggiornando tutte le strutture dati interne per la gestione efficiente delle connessioni. A questo punto solleva un evento che sarà gestito dal Listener registrato attraverso il metodo **setAcceptHandler**.

Connect L'evento connect si verifica quando una connessione in corso è terminata con successo oppure è fallita. In caso di successo il ChannelManager crea una nuova connessione aggiornando le strutture dati interne e solleva l'evento **connectionEstablished**. In caso di fallimento, se sono stati effettuati tutti i tentativi previsti si considera fallita la connessione sollevando l'evento **connectionFailed** altrimenti si registra l'esecuzione di una nuova connessione dopo un secondo. La gestione dei timer ha creato dei problemi durante lo sviluppo. Non è stato possibile utilizzare un timer esterno nel quale registrare tutte le connessioni in quanto un timer deve essere implementato estendendo un nuovo thread e l'oggetto Selector richiede che tutti i comandi gli siano forniti dallo stesso thread nel quale è in esecuzione. Il problema è stato risolto implementando una "TimeLine"

nello stesso oggetto ChannelManager e gestendo gli "sleep" limitando le attese sul metodo **select** del Selector.

Write Con il metodo **sendTCPMessage** o **sendUDPMessage** non si invia direttamente il messaggio ma lo si salva su una lista. L'evento write ci comunica che la socket è disponibile per la scrittura e quindi possiamo inviare i messaggi pendenti. L'invio di un messaggio più lungo della lunghezza di un pacchetto TCP o UDP comporta la frammentazione dello stesso con la consequente difficoltà nella ricomposizione da parte del destinatario. Per questo motivo, prima ogni messaggio si invia una sequenza di quattro byte che codifica la lunghezza del messaggio che segue.

Read L'evento read è sollevato quando l'altro peer ha chiuso la connessione oppure è arrivato un pacchetto dati su una delle socket registrate. Nel primo caso si aggiornano le strutture dati per gestire le connessioni e si solleva l'evento **connectionEnded** mentre nel secondo caso la gestione è un po' più complicata in quanto bisogna gestire anche i messaggi più lunghi. Ricordiamo che all'inizio di ogni messaggio sono inviati 4 byte dove è codificata la lunghezza del messaggio stesso. In questo modo possiamo sempre sapere quanti byte bisogna ancora ricevere per completare il messaggio. Per ogni connessione aperta si mantiene una stringa che mantiene il messaggio incompleto e, ogni volta che si riceve un nuovo pacchetto, si concatena la stringa ricevuta. Al compimento dell'intero messaggio si crea l'evento **messageReceived** che verrà intercettato dal Listener registrato.

Grazie a tutte queste procedure il ChannelManager riesce a creare un livello di astrazione tale da nascondere tutte le problematiche legate all'implementazione delle socket e la frammentazione dei messaggi creando una corrispondenza diretta tra il metodo **sendMessage** e l'evento **messageReceived**.

6.9 OutgoingMessageManager

L'OutgoingMessageManager è il componente che si occupa dell'impacchettamento dei messaggi da inviare. Ogni pacchetto consiste in un messaggio JSON nel quale sono specificati:

- Il tipo del messaggio;
- L'ID del mittente;
- L'ID del destinatario;
- Un campo contenente le informazioni accessorie dipendenti dal tipo di messaggio;
- Il payload contenente il messaggio del livello applicativo inviato dal servizio.

Questo componente mette a disposizione una serie di metodi, uno per ogni tipo di messaggio, attraverso il quale riceve le informazioni necessarie per la creazione della stringa Json che invierà al ChannelManager.

6.10 ReceivedMessageDispatcher

É il componente che si occupa della distribuzione al corretto destinatario di tutti i messaggi ricevuti. Visto che si riceve una stringa codificata in JSON, il primo passo consiste nel parsing della stringa stessa. A questo punto si controlla se il nodo è il reale destinatario del messaggio. Questa situazione è possibile in quanto, in base a quanto osservato in precedenza, un qualsiasi peer può avere la funzione di buddy server di un alto peer. Infatti, se il nodo risulta essere il buddy del destinatario descritto nel messaggio, questo lo si inoltra al destinatario corretto altrimenti si risponde al mittente inviando un messaggio di errore.

Se invece il messaggio è indirizzato proprio al nodo in questione, si controlla l'ID del mittente. Se esiste già un oggetto Node che si occupa della gestione

della comunicazione con il mittente si inoltra il messaggio a questo oggetto altrimenti si inoltra al CommunicationManager al quale spetta il compito della creazione di un nuovo oggetto Node.

6.11 CommunicationManager

Il CommunicationManager è il componente che "fa da cuscinetto" tra la libreria sviluppata e il codice applicativo. L'astrazione offerta al servizio nasconde tutti i dettagli dovuti alle comunicazioni di rete e si fornisce un'interfaccia molto semplice ed essenziale. Il servizio, infatti può solamente inviare un messaggio affidabile o non affidabile ad un determinato nodo fornendo solamente il certificato che lo identifica e può ricevere un evento di messaggio ricevuto da un nodo identificato da un certificato.

Il CommunicationManager si occupa di gestire i messaggi ricevuti da peer non ancora conosciuti. Come accennato in precedenza, il primo messaggio che un nodo invia per stabilire la connessione è di tipo **connection** e contiene il proprio certificato per presentarsi con all'altro peer. Grazie alle informazioni contenute in questo certificato, il CommunicationManager crea un nuovo oggetto Node che si occuperà di stabilire una nuova sessione sicura con il peer. Tutti gli altri tipi di messaggi ricevuti sono gestiti rispondendo con un messaggio di errore.

6.12 Node

Il Node è il componente centrale di tutta l'architettura. Il suo compito è quello di implementare il protocollo che consente la creazione di un tunnel cifrato per la comunicazione tra due peer.

Il componente affronta essenzialmente due problemi:

- la creazione di una connessione diretta che superi la presenza dei router NAT attraverso il supporto dei buddy server;

- la messa in sicurezza della connessione di cui sopra cifrando il contenuto con una chiave di sessione generata attraverso il protocollo di Diffie-Hellman

Vediamo ora in dettaglio come vengono affrontati e risolti i problemi elencati. Durante la discussione ci concentreremo soltanto sui messaggi affidabili ovvero quelli che richiedono l'utilizzo di una socket TCP. Per i messaggi non affidabili il funzionamento è analogo tranne per il fatto che non si deve gestire la connessione.

6.12.1 La creazione di una connessione

Una nuova connessione si crea quando l'applicazione chiede al CommunicationManager di inviare un messaggio ad un determinato peer. Il CommunicationManager controlla se il peer è già gestito da un oggetto Node e in caso contrario crea un nuovo Node fornendogli tutte le informazioni elencate nel certificato. Consideriamo il secondo caso: a questo nuovo oggetto si assegna la gestione di tutta la comunicazione con il peer e gli si inoltra il comando di **sendMessage** specificando un Listener per gli eventi causati dal fallimento dell'invio del messaggio.

Sappiamo che non si è ancora inviato e ricevuto nessun messaggio con il peer altrimenti l'oggetto Node sarebbe già esistente. Per inviare un nuovo messaggio dobbiamo prima stabilire una connessione inviando un messaggio **connect** al ChannelManager e registrando un Listener per gli eventi di connessione. Se si riceve un evento di connessione fallita si crea un evento di **messageNotSent** che viene gestito dal Listener specifico. Se invece la connessione va a buon fine si invia un messaggio di **connection** allegando il proprio certificato. Sul peer destinatario probabilmente non esiste ancora un oggetto Node per la comunicazione con il nodo quindi il messaggio di **connection** verrà inoltrato al CommunicationManager che creerà un nuovo Node inizializzandolo con le informazioni descritte nel certificato ricevuto.

Nella situazione ideale, dato che la connessione è ormai stabilita, i due peer potrebbero eseguire immediatamente il protocollo per la generazione di una chiave di sessione e avviare così la comunicazione.

Purtroppo la situazione non è sempre così semplice a causa della presenza di NAT che saranno presenti nella rete finché il protocollo IPv6 non sarà operativo. Bisogna considerare alcuni casi particolari descritti di seguito.

Il nodo che si vuole contattare è a monte di un router NAT

Per contattare questo peer è necessario aprire una connessione con il buddy descritto nel certificato del peer stesso. Una volta contattato il buddy, questo inoltrerà il messaggio di **connection** con certificato allegato al destinatario desiderato il quale, verrà a conoscenza delle informazioni riguardanti il mittente. Se si scopre che il mittente è accessibile dalla rete, si apre una nuova connessione diretta e si chiude la vecchia implementando così il protocollo di *connection reversal*.

I due nodi si trovano nella stessa rete locale

Entrambi i nodi sono dietro un NAT e potrebbero non poter attivare il PortMapping per essere accessibili dall'esterno. In questo caso non vorremmo inoltrare tutti i nostri messaggi verso un buddy server da qualche parte sulla rete ma vorremmo che i peer comunicassero sfruttando solamente la rete locale. Per individuare questo caso, la soluzione più semplice consiste nel confrontare le informazioni dei certificati. Se entrambi i nodi sono nascosti da un NAT, non sono accessibili e hanno lo stesso indirizzo IP pubblico, vuol dire che si trovano nella stessa rete locale. C'è però ancora un caso nel quale due nodi, nonostante appartengano alla stessa rete locale, non hanno lo stesso indirizzo IP pubblico. Infatti, se la rete locale ha due diversi Gateway, i due peer possono uscire dalla rete con due indirizzi IP diversi e venire considerati come appartenenti a due reti diverse. L'unica soluzione per comprendere tutti i casi consiste nel tentare sempre due connessioni, una sulla rete locale e l'altra sulla rete pubblica o buddy. Se entrambe le connessioni vanno a buon fine si mantiene sempre

quella diretta attraverso la rete locale. Può anche capitare che si vuole contattare un nodo su un'altra rete che ha lo stesso indirizzo locale di un altro nodo nella nostra rete locale. In questo caso il peer sulla nostra stessa rete locale viene contattato per sbaglio, quando riscontra che il messaggio non è indirizzato a lui risponde con un errore.

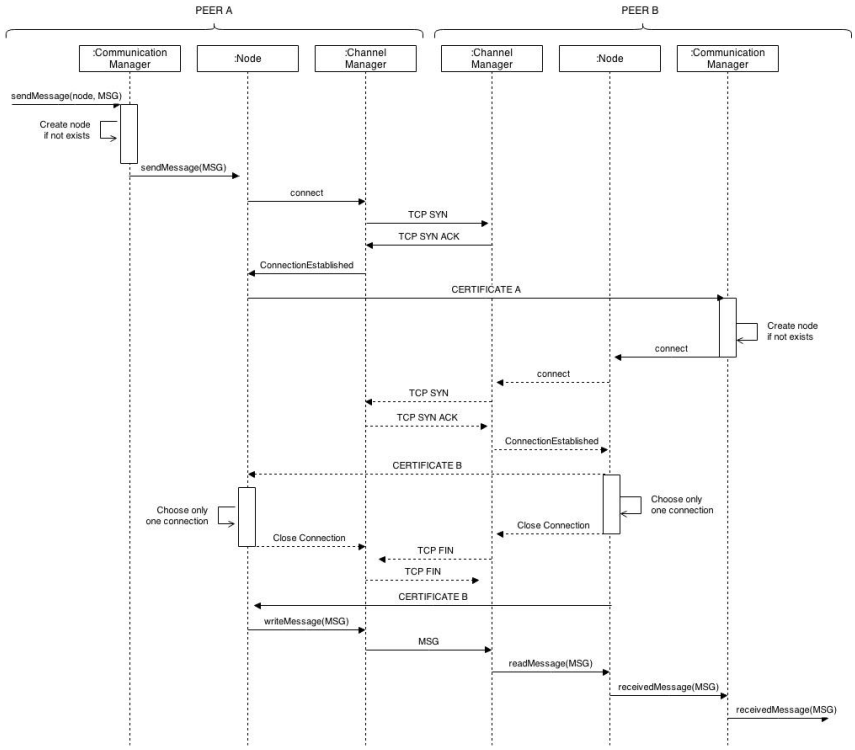
I due nodi si trovano su due livelli diversi della stessa rete locale

Questo caso lo possiamo vedere come l'unione dei due visti precedentemente. Immaginiamo infatti che due peer siano sulla stessa rete locale ma uno dei due è nascosto da un altro router NAT. Per coprire anche questo caso bisogna tentare la connessione verso entrambi gli indirizzi pubblico e privato. Il peer che riceve un messaggio di **connection** deve a sua volta tentare una nuova connessione verso entrambi gli indirizzi al mittente.

Il protocollo derivante dall'unione delle soluzioni appena presentate e mostrato nel diagramma in figura 6.6 riesce a far stabilire una connessione diretta nella maggior parte delle circostanze che possono accadere su internet. Nel diagramma è mostrata una sola connessione verso l'altro peer e non si evidenzia l'esistenza di due connessioni verso l'indirizzo locale e pubblico. Ipotizziamo che quella riportata sia l'unica connessione tra le due che va a buon fine oppure, in caso di successo da parte di entrambe, si immagini che quella verso l'indirizzo pubblico sia già stata chiusa. Come possiamo vedere, secondo il protocollo si creano sempre due connessioni: una creata dal peer A i cui messaggi sono evidenziati con la linea unita e l'altra dal peer B i cui messaggi sono identificarti da linee tratteggiate. Entrambi i nodi inviano, come primo messaggio il proprio certificato essendo così sicuri che l'altro peer possa aver istanziato l'oggetto Node per la gestione della comunicazione.

Nell'immagine vi è una parte del protocollo non ancora affrontata ovvero la scelta di una tra le due connessioni esistenti. Entrambi i nodi sono a conoscenza se le connessioni sono state create attraverso una rete locale, una rete pubblica oppure passano attraverso un buddy. Se le due connessioni hanno origine diverse si dà la priorità a quella locale piuttosto che a quella pubblica piuttosto

Figura 6.6: Protocollo semplificato per la comunicazione End-to-End



che a quella attraverso il buddy. Se invece l'origine è la stessa, abbiamo bisogno di una regola che consenta a entrambi i peer di scegliere la stessa connessione evitando altri messaggi che rallenterebbero solamente la procedura.

Le due connessioni formate sono identificate dalle seguenti quadruple

$$< IP_A, porta_{A1}, IP_B, porta_{B1} > \tag{6.1}$$

$$< IP_A, porta_{A2}, IP_B, porta_{B2} > \tag{6.2}$$

La procedura per scegliere una connessione consiste nel creare quattro stringhe ognuna della quali è formata da $IP_X, porta_{Xj}$. Si ordinano lessicograficamente le quattro stringhe si sceglie la minore e si mantiene la connessione contenente la coppia $< IP_X, porta_{Xj} >$ che ha generato la stringa scelta. L'altra si chiude.

Nonostante l'abbondanza di situazioni descritte, ne esiste ancora un'altra che discuteremo a parte perché la cui gestione porta l'unica distinzione nell'implementazione tra i messaggi TCP e UDP. Stiamo parlando del caso in cui

i due nodi si trovano su due reti locali diverse create da router che non supportano il PortMapping. L'unica soluzione che rimane da provare per stabilire una connessione diretta è l'utilizzo dell'hole punching[17]. L'implementazione del protocollo richiede di lavorare **solo** su due ricevitori: uno per tutti i messaggi TCP e uno per tutti i messaggi UDP. Mentre UDP non è orientato alla connessione, tutti i messaggi arrivano su un unica socket ed è così possibile implementare l'hole punching. Nel caso di TCP invece, i messaggi che arrivano sono distribuiti su più socket, una per ogni connessione. Questo rende impossibile l'implementazione dell'hole punching lavorando ad alto livello. É possibile scendere di un livello e implementare il protocollo a livello IP riscrivendo però tutte le funzalità di TCP.

Dato che si sta implementando un prototipo e l'eventualità che si verifichi una situazione simile è molto bassa, non si è ritenuto necessario scendere così a basso livello ed è stato implementato l'hole punching solo per lo scambio di messaggi UDP. Nel caso di connessioni TCP si richiederà l'aiuto di un buddy.

Questo, nonostante inoltrerà tutti i messaggi, non riuscirà ad intercettare niente grazie alla cifratura end-to-end con firma della chiavi pubbliche descritta qui di seguito.

6.12.2 La messa in sicurezza della connessione creata

Il problema della creazione di un tunnel sicuro dove e peer possano comunicare in tutta sicurezza è già stato affrontato più volte in questa trattazione approfondendolo a diversi livelli. In questo paragrafo ci dedicheremo all'integrazione del protocollo di Diffie-Hellman con quello appena descritto.

La buona notizia è che i due protocolli si integrano abbastanza facilmente dato che entrambi prevedono un handshake formato da due messaggi. Il protocollo descritto prevede lo scambio dei certificati mentre Diffie-Hellman prevede l'invio delle chiavi pubbliche.

Un peer, quindi, prima di creare una nuova connessione, genera i parametri necessari per la creazione della coppia di chiavi di Diffie-Hellman e la coppia di chiavi stessa. Una volta stabilita la connessione, il primo messaggio di tipo

connection conterrà, oltre al certificato, anche una stringa firmata contenente i parametri di Diffie-Hellman e la chiave pubblica.

Il peer ricevente, attraverso la chiave pubblica RSA contenuta nel certificato potrà verificare la firma con la quale si rendono autentici i parametri e la chiave pubblica di DH. Partendo dai parametri ricevuti anche questo peer si calcolerà la propria coppia di chiavi DH e invierà anche lui la propria chiave pubblica firmata insieme al certificato. Grazie a questi passaggi entrambi i nodi possono generarsi la stessa chiave di sessione per cifrare i successivi messaggi.

La cifratura del canale avviene utilizzando uno dei più sicuri cifrari simmetrici al momento esistenti: AES-256. Come si è spiegato parlando della crittografia, per rendere più sicuro un cifrario simmetrico a blocchi come AES con messaggi lunghi, è necessario utilizzare la tecnica della composizione di blocchi. Questa tecnica richiede appunto l'inserimento di un *initial vector* che deve essere uguale durante la cifratura e decifratura.

Anche in questo caso si vuole evitare l'invio di messaggi inutili e si è quindi deciso di generare un vettore casuale. Per fare in modo che il vettore sia lo stesso per entrambi i peer, si definisce lo stesso seme per il generatore casuale calcolandolo in funzione della chiave di sessione.

Nel caso di messaggi corti e ripetuti, il protocollo potrebbe essere soggetto ad attacchi sulla chiave quindi, per evitare ciò si è deciso di inizializzare l'*initial vector* di AES ad ogni messaggio. Ogni peer mantiene il numero di messaggi inviati e ricevuti e si utilizza questo numero, insieme alla chiave di sessione per la generazione di nuovi semi e quindi *initial vector* sempre diversi. Dato che il numero di messaggi inviati di un peer è lo stesso di quelli ricevuti dall'altro, la cifratura e decifratura sono sempre possibili. Inoltre, con questa tecnica ci si protegge anche da un attaccante che copia i pacchetti sulla rete e le reinvia più volte al destinatario corretto. In questo modo il primo pacchetto verrà decifrato correttamente mentre per gli altri non varrà più l'equivalenza tra il numero di messaggio inviati e ricevuti comportando l'errata decifratura con consequente eliminazione del pacchetto.

6.13 Un caso d'uso: la Chat

L'architettura e i protocolli presentati in questo capitolo consentono la creazione di un tunnel per una comunicazione sicura ed efficiente tra peer. La libreria presentata è stata sviluppata da poco ed ha ancora bisogno di essere testata. Intanto per mostrare la sua utilità e semplificare l'alpha testing è stato creato un servizio di chat molto semplice e minimale.

Nell'architettura descritta in 6.7, la chat si posiziona al livello applicativo e interagisce direttamente con il CommunicationManager. Proprio per questo motivo, in base alle specifiche del CommunicationManager spiegate in precedenza, per inviare un messaggio ad un certo nodo N, bisogna essere in possesso del certificato del nodo N così da conoscere tutte le informazioni per stabilire una connessione.

L'applicazione chat ha quindi bisogno di un servizio che mantenga aggiornata una lista di tutti gli utenti con i quali l'utilizzatore della chat può interagire con il relativo stato e, in caso di nodo *online*, anche il certificato. Questo servizio di gestione della chat, che può essere situato anche su un server indipendente, deve mantenere l'elenco di tutti i certificati degli utenti connessi. Per fare questo, ogni utente che vuole sfruttare il servizio di chat, dovrà connettersi al gestore della chat stessa e inviargli il proprio certificato. Il server, per essere sicuro che il client che si connette è il reale proprietario del certificato, invia al peer una stringa casuale chiamata *nonce* che il client dovrà rispedire firmata. La firma, che può essere verificata soltanto con i dati memorizzati nel certificato, indica che il client è in possesso della chiave privata e quindi l'utente è verificato.

Dato che l'utente può trovarsi in più stati (*online*, *offline*, *busy*, ecc), il codice di questo stato viene inviato al server e firmato insieme al *nonce*. In questo modo il server ha una panoramica completa dello stato degli utenti e, appena riceve una modifica, invia le variazioni a tutti gli altri client interessati.

Grazie al gestore della chat, il client può essere sempre aggiornato sullo stato dei contatti dell'utente e può mostrare queste informazioni all'utente

stesso tramite un'interfaccia grafica.

L'implementazione dell'interfaccia utente non è stata implementata secondo le consuete tecniche previste da Java come AWT[5] o Swing[7] ma, grazie alla possibilità di includere un web browser nell'interfaccia grafica della nostra applicazione, è stato ritenuto più opportuno implementare la grafica come un sito web scritto in HTML[41], CSS[39] e Javascript[12]. La scelta è stata dovuta principalmente alla maggiore flessibilità e semplicità di HTML rispetto a Java nella programmazione grafica.

Grazie a questa chat due utenti possono comunicare, inviarsi password o pin della carta di credito essendo sicuri che nessuno è in grado di decifrarli.

6.14 L'analisi delle latenze

Come ultima fase del lavoro presentato si è deciso di fare una breve analisi dei ritardi derivanti dall'utilizzo della libreria descritta. Per calcolare le latenze di comunicazione è stato creato un servizio che può comportarsi sia come client che rileva le latenze, sia come server echo. Il servizio lato client invia un messaggio e attiva un timer. Quello lato server si comporta come eco quindi, alla ricezione di un messaggio, risponde al mittente inviando lo stesso messaggio ricevuto. Una volta ricevuto il messaggio di eco, il servizio lato client calcola il tempo intercorso tra l'invio del messaggio e la ricezione dell'eco e, dividendo per due questo tempo, può stimare il tempo impiegato per l'invio di un solo messaggio.

Per stimare il ritardo provocato dalla libreria è stato confrontata la latenza registrata dal servizio appena descritto con la latenza di rete calcolata dal comando **ping** disponibile sia su macchine Windows che su macchine derivanti da sistemi Unix.

Nella tabella seguente riporteremo per ogni riga una media di 20 latenze calcolate in diverse ore della giornata tra due peer situati entrambi a Pisa.

Ping del sistema operativo	39 ms
Invio dei un messaggio TCP "in chiaro"	52 ms
Invio dei un messaggio TCP cifrato con AES-256	58 ms
Invio del primo messaggio TCP ad un peer accessibile	2.384 ms
Invio del primo messaggio TCP ad un peer NATTATO	5.253 ms

Dalle misure ottenute si può verificare come la libreria, nonostante la sua architettura, abbia un impatto quasi nullo sulle latenze. Calcolando infatti la differenza tra i valori delle latenze tra l'invio di un messaggio in chiaro e quello di un ping di sistema, possiamo verificare che, in media, la libreria comporta un ritardo di circa 13ms che è praticamente impercettibile. Inoltre i dati ci mostrano anche quanto la cifratura simmetrica con AES-256 sia efficiente. La cifratura di un messaggio comporta degli ulteriori ritardi di circa 6 ms. Queste informazioni ci mostrano quanto la libreria sia efficente "a regime" ovvero quando la connessione è già stata stabilita.

Proprio la stabilizzazione di una connessione, lo scambio delle chiavi durante l'handshake tra nodi e soprattutto le verifiche delle firme dei certificati e chiavi pubbliche di Diffie-Hellman comportano dei ritardi notevoli che variano a seconda della situazione dei nodi. Se il nodo che si vuole contattare è situato a monte di un NAT, bisogna sfruttare il supporto di un buddy server e implementare il protocollo di connection reversal. Per come è progettato il protocollo implementato, in un primo momento si stabilisce una connessione attraverso un buddy mentre solo in un secondo momento si stabilisce una seconda connessione diretta tra i due nodi sulla quale vengono inviati i messaggi. In base ai dati raccolti, infatti, si può notare come l'invio del primo messaggio ad un nodo NATTATO comporti dei ritardi maggiori di circa il 220% rispetto al caso nel quale il nodo destinatario fosse accessibile. Questo maggiore ritardo è dovuto all'esecuzione di due handshake su entrambe le connessioni e dalle maggiori latenze della connessione effettuata attraverso il buddy.

Infine bisogna notare che il protocollo implementato comporta queste latenze così elevate soltanto per l'invio del primo messaggio. Per il resto della comunicazione il ritardo è praticamente impercettibile. Quindi nel caso in cui

si voglia inviare un unico messaggio di chat il costo dell'handshake influisce notevolmente sulle latenze medie mentre, se si vuole inviare un file abbastanza grande, il costo dell'handshake viene "spalmato" sui diversi pacchetti inviati comportando un ritardo medio quasi nullo.

Capitolo 7

Considerazioni conclusive

Al giorno d'oggi, la maggior parte delle persone fa uso, consapevolmente o inconsapevolmente di servizi cloud che offrono sistemi di elaborazione o storage ad un prezzo proporzionale al loro reale utilizzo. Questo permette l'evitare di investire ingenti somme per l'acquisizione di grossi sistemi di elaborazione in grado di supportare un carico di lavoro elevato, quando in realtà l'utilizzo delle risorse acquistate è molto inferiore in media, mentre, si verificano dei poi ma si toccano dei picchi di utilizzo solamente in determinati periodi di tempo. Grazie alla tecnologia Cloud è possibile "affittare" delle risorse in più soltanto quando servono permettendo al gestore dei servizi dei grossi risparmi. Uno dei maggiori problemi dei servizi cloud consiste nella sicurezza delle informazioni infatti, i gestori cloud hanno la possibilità di accedere ad una vasta mole di dati che potrebbero essere letti ed analizzati senza il consenso dei proprietari dei dati stessi. Quindi, superata una prima fase di "euforia", molte aziende stanno decidendo di tornare alla vecchia tecnologia proprio per garantire la riservatezza delle proprie informazioni.

In questo contesto entra in gioco la nuova architettura proposta in questa tesi e progettata presso l'azienda eLearnSecurity con l'ambizione di creare un'infrastruttura che consenta la gestione di più **servizi cloud** in un'**unica** piattaforma in totale sicurezza: Izzie. Per limitare i costi, soprattutto dovuti alla banda, del cloud, Izzie si integra con una rete P2P che possa creare un

tunnel diretto e sicuro tra due dispositivi che vogliano comunicare e una DHT anch'essa sicura che può essere usata come supporto di alcuni servizi.

Dato che molte aziende non intendono di salvare dati, anche cifrati, su una DHT per il fatto che una rete P2P non consente di decidere dove queste informazioni vengano salvate, la DHT progettata prevede l'introduzione del concetto di gruppo cioè un insieme di peer utilizzati da persone autorizzate. Un documento salvato sulla DHT, può essere vincolato ad essere salvato solamente sui peer utilizzati da utenti facenti parte di un precisato gruppo. Grazie a questa nuova tecnica, un'azienda può creare e gestire il gruppo dei propri dipendenti e utilizzare soltanto i dispositivi di queste persone, per mantenere i propri dati che sono comunque cifrati.

In questa tesi ci si è concentrati sulla rete P2P sicura integrata con l'architettura Izzie. Si è studiato, infatti, in un primo momento la struttura e il funzionamento di una DHT considerando come caso di studio la rete Pastry. É stato visto che le DHT sono nate per la condivisione di contenuti e quindi, nelle reti esistenti, non sono stati previsti sistemi di sicurezza. Questo fatto le rende quindi vulnerabili ad una serie di attacchi che un malintenzionato può effettuare. Questi attacchi sono stati analizzati e, sulla base di soluzioni proposte in letteratura, è stata progettato un protocollo distribuito che riesca ad eliminare tutti gli attacchi studiati che costituiscono la parte più importante di quelli esistenti. Tutto questo è stato possibile grazie all'integrazione con i sistemi di sicurezza forniti dall'architettura Izzie.

Dopo la fase di progettazione si è passati all'implementazione della DHT seguendo un approccio *bottom-up* ovvero sviluppando come prima obiettivo una libreria per la comunicazione sicura tra peer con cifratura end-to-end.

É stato inoltre sviluppato un servizio di chat che consente a due utenti, interni all'azienda eLearnSecurity, di comunicare tra loro in tutta sicurezza avendo quindi la possibilità di inviarsi password o informazioni sensibili essendo sicuri che nessuno riesca ad intercettare e leggere i messaggi.

7.1 Sviluppi futuri

Il progetto presentato è ancora in via di sviluppo. La parte della comunicazione end-to-end e l'integrazione con i blocchi è stata sviluppata sotto forma di libreria ed è stato implementato un servizio di chat come caso d'uso della libreria stessa. É in programma il completamento dello sviluppo della DHT con la gestione dei gruppi per implementare un servizio di storage distribuito già progettato e descritto nel capitolo 5. In parallelo allo sviluppo della DHT si prevede di finire l'implementazione dei blocchi di controllo di tutta l'infrastruttura Izzie e dei servizi di chat. Inoltre è previsto lo sviluppo di un Authentication Provider proprietario e l'estensione dell'insieme di quelli supportati. Attualmente infatti sono supportati solamente Google e Facebook e si vorrebbe estendere l'insieme implementando anche il supporto a Twitter e Linkedin.

Una volta implementata l'infrastruttura completa e verificato il corretto funzionamento, di prevede di realizzare un insieme di ottimizzazioni che fino a questo momento non sono state prese in considerazione, dato che è stato progettato un prototipo dell'architettura.

Oltre alle ottimizzazioni è necessario, sulla base del servizio di chat aziendale creato, creare un nuovo servizio che permetta l'utilizzo della chat anche alle persone esterne all'azienda. La chat dovrà essere estesa permettendo in un primo momento l'invio di file in maniera sicura e successivamente la possibilità di effettuare videochiamate. Infine si prevede di implementare un servizio di storage distribuito che possa sfruttare le potenzialità dei gruppi della DHT per permettere alle aziende un migliore sfruttamento delle risorse interne.

7.1.1 Ottimizzazioni

Le ottimizzazioni da implementare che descriveremo in questa sezione riguardano solamente la parte implementata fino a questo momento. Le principali ottimizzazioni che si prevede di implementare sono le seguenti.

Riscrittura del ChannelManager utilizzando un linguaggio a basso livello come C o C++ che ci consenta di accedere al livello IP dello stack di rete per implementare funzioni come l'hole punching su protocollo TCP. In questo modo è possibile ottenere una connessione punto-punto tra due nodi nascosti da un NAT evitando inutili passaggi attraverso un nodo buddy.

Introduzione di un header a lunghezza fissa nei messaggi scambiati tra due nodi della rete P2P. Nel prototipo i messaggi scambiati sono in formato Json che permette un'alta flessibilità ma comporta un aumento della lunghezza dei messaggi. Basti pensare che per salvare un valore booleano, con la giusta codifica è necessario un bit mentre attraverso un messaggio Json è necessaria una stringa come "{*nome campo:(true-false)*}". Mentre per l'invio di messaggi brevi questo incremento di spazio comporta un overhead irrilevante, questo overhead si accumula nel caso di invio di messaggi più lunghi provocando ritardi percettibili. Modificando la codifica Json con un header dove le informazioni sono codificate in modo da rendere minimo lo spreco di spazio, si riuscirebbe ad ottenere una diminuzione del numero di pacchetti necessari per lo scambio di messaggi lunghi con conseguente miglioramento delle performance.

Riscrittura dell'interfaccia che collega il CommunicationManager con i servizi in modo tale da trasformare un servizio in un *plugin* che possa essere scritto da chiunque e riesca a sfruttare a pieno le potenzialità dell'architettura sottostante.

7.1.2 Le estensioni della chat

Attualmente la chat permette solo l'invio di messaggi tra gli utenti interni all'azienda. Il primo obiettivo da raggiungere consiste nel permettere l'utilizzo del servizio a tutti gli utenti. Ricordiamo che, attualmente, il gestore di chat mantiene tutti i certificati dei nodi connessi e lo stato di tutti gli utenti che possono accedere, risultando un collo di bottiglia e single-point-of-failure per

tutto il servizio. Per estendere il bacino di utenza del servizio di chat è necessaria una gestione distribuita degli utenti e dei loro certificati sfruttando le potenzialità della DHT.

La chat dovrà essere estesa anche dal punto di vista funzionale aggiungendo la possibilità di inviare file e di effettuare videochiamate. Mentre si prevede che l'implementazione delle videochiamate richieda un notevole lavoro, l'invio di file non dovrebbe creare grossi problemi dal punto di vista tecnico perché la libreria implementata permette di inviare anche file molto grandi.

7.1.3 Un servizio storage distribuito

Molte aziende non hanno mai usato servizi di storage distribuito su rete P2P, perché non è garantito nessun sistema di sicurezza. Preferiscono invece servizi di storage cloud nonostante non garantiscano comunque un alto livello di riservatezza.

L'osservazione fondamentale è che alcune grandi aziende sottoutilizzano alcune delle risorse interne, ad esempio lo storage. Ipotizziamo che in un ufficio siano presenti 100 computer che montano dischi da 100 Gigabyte utilizzati in una minima parte: 30 Gigabyte. Se riuscissimo ad utilizzare 50 Gigabyte per ognuno dei 100 computer, ci troveremmo circa 5 Terabyte di spazio già pagato che potremmo utilizzare per memorizzare i documenti aziendali, o perlomeno quelli più sensibili, senza ricorrere ad un servizio di storage su cloud. Questa soluzione, da un punto di vista tecnico, aumenta la velocità nel reperire i dati visto che questi sono già sulla rete locale e, da un punto di vista commerciale, ci consente una diminuzione dei costi dovuti appunto allo storage e alla banda di rete utilizzata.

Si prevede, quindi, la creazione di un servizio integrato con l'architettura Izzie, che utilizzi i gruppi della DHT per implementare uno storage interno all'azienda che risulti economico e affidabile. Mentre si è già discusso dell'impatto economico che un servizio di questo tipo possa avere, l'affidabilità è data dall'organizzazione della DHT che mantiene le repliche di tutti i dati evitando che il fallimento di un computer possa provocare qualche perdita dei dati.

Ringraziamenti

Fino a questo momento della mia vita ho semplicemente seguito delle linee guida già definite per la mia carriera scolastica e accademica. Infatti, sin da bambino, avevo deciso che avrei studiato ragioneria, al terzo anno avrei preso l'indirizzo per programmatori e avrei preso una laurea in informatica a Pisa. La tesi che ho presentato rappresenta per me un punto di rottura con il passato. Non ho più un programma da seguire ma dovrò valutare varie ipotesi che mi si presenteranno davanti e decidere un nuovo programma di vita. Guardando indietro mi sento il dovere di fare dei ringraziamenti.

Prima di tutti un ringraziamento particolare va alla professoressa Ricci per la sua disponibilità dimostrata durante la stesura di queste pagine.

Vorrei anche ringraziare Andrea Tarquini per avermi voluto al suo fianco nella progettazione e sviluppo del progetto Izzie e per avermi sempre dato degli utili consigli figli dell'esperienza accumulata in campo lavorativo. Un grazie sincero anche a Emanuele Simonelli per avermi messo a disposizione il suo computer da utilizzare per i test sulle comunicazioni di rete e per esserci sempre stato nei momenti di bisogno.

Un ringraziamento va a tutte le persone (non li nomino per paura di dimenticarmene qualcuno) che mi sono state vicine in questi anni di vita pisana. Un pensiero va a tutti i miei coinquilini, colleghi e amici che mi hanno aiutato dandomi utili consigli oppure facendomi passare qualche bella serata in compagnia.

Vorrei ringraziare tutte le persone che mi hanno sopportato tutti i giorni ma, soprattutto, tutte quelli che, invidiosi dei miei successi, hanno sempre

goduto vedendomi inciampare nel mio percorso. É proprio grazie a loro che ho sempre trovato la forza per rialzarmi ed affrontare al 110% tutti problemi che mi si ponevano davanti.

Ovviamente non posso non dire un grazie speciale alla mia famiglia. Il supporto morale ed economico è stato fondamentale negli ultimi anni universitari ma soprattutto nella vita. Un grazie per avermi insegnato a rispettare le persone senza farsi mettere i piedi in testa. Un grazie per avermi insegnato ad essere sempre serio, preciso ed educato nei rapporti con gli altri. Un grazie per avermi insegnato la cultura del lavoro e del sacrificio perché senza questi componenti non si raggiungono dei traguardi importanti nell'università, nel lavoro e, soprattutto, nella vita. Un grazie per avermi insegnato ad essere quello che sono.

Elenco delle figure

Bibliografia

[1] R. A. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 312–320, New York, NY, USA, 2005. ACM.

[2] N. Borisov. Computational puzzles as sybil defenses. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, P2P '06, pages 171–176, Washington, DC, USA, 2006. IEEE Computer Society.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, Dec. 2002.

[4] S. Cheshire, M. Krochmal, and A. Inc. Nat port mapping protocol. `https://tools.ietf.org/html/rfc6886`, April 2013. RFC 6886.

[5] O. Corporation. Package java.awt. `https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html`.

[6] O. Corporation. Package java.nio. `https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html`.

[7] O. Corporation. Trail: Creating a gui with jfc/swing. `https://docs.oracle.com/javase/tutorial/uiswing/`.

[8] D. Crockford. Json, the fat-free alternative to xml. *In Proceeding of XML 2006 conference. Boston*, 2006.

[9] N. Datta. Zero knowledge password authentication protocol. *International Journal of Communication Network Security*, 2012.

[10] J. Dinger and H. Hartenstein. Defending the sybil attack in p2p networks: Taxonomy, challenges, and a proposal for self-registration. In *Proceedings of the First International Conference on Availability, Reliability and Security*, ARES '06, pages 756–763, Washington, DC, USA, 2006. IEEE Computer Society.

[11] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 75–, Washington, DC, USA, 2001. IEEE Computer Society.

[12] ECMA. Standard 262 ecmascript language specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

[13] ECMA. Introducing json. `http://www.json.org/`, Dec. 1999.

[14] eLearnSecurity srl. Forging security professionals. `www.elearnsecurity.com/`.

[15] P. Ferragina and F. Luccio. *Crittografia*. Bollati Boringhieri, 2007.

[16] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[17] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[18] W. Foundation. Wireshark. `https://www.wireshark.org/`.

[19] E. Hammer. Oauth 1.0. `https://tools.ietf.org/html/rfc5849`, April 2010. RFC 5849.

[20] D. Hardt. Oauth 2.0. `https://tools.ietf.org/html/rfc6749`, October 2012. RFC 6749.

[21] M. E. Hellman. An overview of public key cryptography. *Comm. Mag.*, 40(5):42–49, May 2002.

[22] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. *In Proceedings of the 17th International Symposium on Distributed Computing*, 2003.

[23] B. Inc. The original bittorrent client. `http://www.bittorrent.com/`.

[24] G. Inc. Google accounts authentication and authorization. `https://developers.google.com/accounts/docs/AuthSub`.

[25] Y. Inc. Authentication and authorization with yahoo! `https://developer.yahoo.com/auth/`.

[26] M. Jelasity. *Self-organising Software*, chapter Gossip. Springer Berlin Heidelberg, 2011.

[27] P. Ltd. Burp suite, the leading toolkit for web application security testing. `http://portswigger.net/burp/`.

[28] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[29] E. Project. emule. `http://www.emule-project.net/`.

[30] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[31] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[32] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 30–43, London, UK, UK, 2001. Springer-Verlag.

[33] A. B. Shehata and D. Castagna. weupnp. `https://code.google.com/p/weupnp/`.

[34] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 261–269, London, UK, UK, 2002. Springer-Verlag.

[35] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, Feb. 2003.

[36] A. Tarquini and eLearnSecurity srl. Justcrypt: Software-less power-ful encryption for your documents. `https://justcrypt.it/`.

[37] G. Urdaneta, G. Pierre, and M. V. Steen. A survey of dht security techniques. *ACM Comput. Surv.*, 43(2):8:1–8:49, Feb. 2011.

[38] R. Villanueva and M. del Pilar Villamil. Secureroutingdht: A protocol for reliable routing in p2p dht-based systems. *Università di Los Andes, Bogotá, Colombia*, 2012.

[39] W3C. Cascading style sheets home page. `http://www.w3.org/Style/CSS/`.

[40] W3C. Extensible markup language xml. `http://www.w3.org/XML/`.

[41] W3C. Html. `http://www.w3.org/html/`.

[42] W3C. Soap current status. `http://www.w3.org/standards/techs/soap`.

[43] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. *IEEE/ACM Trans. Netw.*, 18(3):885–898, June 2010.

[44] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J.Sel. A. Commun.*, 22(1):41–53, Sept. 2006.