POLITECNICO DI MILANO
FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni
Dipartimento di Elettronica e Informazione

IDENTIFICAZIONE E DIFESA CONTRO L'ATTACCO ECLIPSE AL PROTOCOLLO CHORD

Relatore: Prof. Giacomo VERTICALE
Correlatore: Ing. Cristina ROTTONDI

Tesi di laurea di:

Alessandro PANZERI Matr. 762690
Constantin T. YAGNE Matr. 771245

ANNO ACCADEMICO 2012 / 2013

Indice

Abstract

Overlay peer to peer networks are becoming more and more subject of study since they seem the ideal solution for providing large scale services instead of the traditional low scalable client-server model. In particular, completely distributed and structured peer to peer overlay networks such as Distributed Hash Table (DHT), have raised big interest for their scalability higher than any model which includes forms of centralization. The most common way to realize these networks is the Distributed Hash Table (DHT) and Chord protocol is one of the first DHT proposed. Not only DHT but more in general distributed overlay networks, being open and self organizing without central authority, imply serious concerns about security since, in theory, each participant is free to act maliciously. One of the most effective attacks on distributed overlay networks is the Eclipse attack, in which the attacker forces the victim nodes point to as many attacker's nodes as possible. As a consequence, the victim nodes will involuntarily misroute the traffic towards attacker's nodes allowing the attacker to eliminate, read, modify a large percentage of messages, even if the fraction of malicious nodes is very low. And given that distributed overlay networks can support a wide variety of online services and applications, it is of paramount importance to make these protocols secure especially against this kind of attacks.

In this thesis we focus on the Eclipse attack in Chord protocol. In particular our work consists in devising possible solutions for two problems:

1) Identification of the attack: we propose a solution with which a Chord node can detect the Eclipse attack by relying just on local information and possibly on received messages.

2) Defense to the attack: we propose several distributed defenses and one non distributed defense for contrasting as much as possible the effect of the Eclipse attack, which is the capture of messages by the attacker. We mainly focused on distributed defenses in order to preserve the distributed high scalable nature of Chord protocol. Besides, since some our proposed defenses need to know the average internode key distance, we also devised a system for estimating such quantity in presence of unreliable routing information.

Sommario

Le reti overlay peer to peer si stanno diffondendo sempre più in quanto soluzione ideale per fornire servizi di larga scala, in luogo del tradizionale poco scalabile modello client-server. In particolare le reti overlay peer to peer distribuite e strutturate, come quelle realizzate mediante Distributed Hash Tables (DHT), hanno suscitato un notevole interesse per la loro elevata scalabilità. Il protocollo Chord è una delle prime DHT proposte. Non solo le DHT ma più in generale le reti overlay distribuite, essendo aperte, auto-organizzanti senza un'autorità supervisionante, implicano seri problemi di sicurezza in quanto, in teoria, ogni utente ha la libertà di agire in modo maligno. Uno degli attacchi più efficaci nelle reti overlay distribuite è l'attacco Eclipse, in cui l'attaccante fa sì che i nodi vittima puntino il più possibile verso nodi controllati dall'attaccante. La conseguenza è che i nodi vittima dirottano involontariamente il traffico verso i nodi maligni, permettendo all'attaccante di eliminare, leggere o modificare un'elevata percentuale di messaggi anche se la percentuale di nodi maligni è molto bassa. E poiché le reti overlay distribuite possono supportare un'ampia varietà di applicazioni, è di capitale importanza renderle sicure, specialmente contro un attacco così compromettente.
In questa tesi ci focalizziamo sull'attacco Eclipse al protocollo Chord. In particolare, il nostro lavoro consiste nel proporre possibili soluzioni a due problemi:
1) Identificazione dell'attacco: proponiamo una soluzione con cui un nodo Chord può identificare un attacco Eclipse soltanto affidandosi alle informazioni locali ed eventualmente osservando anche i messaggi ricevuti.
2) Difesa contro l'attacco: proponiamo diverse difese distribuite e una difesa non distribuita per contrastare il più possibile l'effetto dell'attacco Eclipse, che è la cattura dei messaggi da parte dell'attaccante. Ci siamo concentrati maggiormente sulle difese distribuite in modo da preservare la natura distribuita altamente scalabile del protocollo Chord. Inoltre, dato che alcune difese proposte necessitano la conoscenza della distanza media fra nodi nello spazio delle chiavi, abbiamo anche ideato un sistema per stimare tale quantità in presenza di informazioni di routing non affidabili.

1
INTRODUZIONE

Le reti overlay peer to peer completamente distribuite stanno attirando un crescente interesse per le loro caretteristiche di robustezza ai guasti e al fatto che non necessitano di un autorità centrale che la gestisca. Con la dicitura rete overlay intendiamo una rete logica costruita sopra la rete fisica: nel nostro ambito una rete overlay è un gruppo di host connessi ad una WAN (ad es. internet) che si organizzano in modo da poter comunicare l'uno con l'altro. Per rete completamente distribuita intendiamo una rete in cui non vi è alcuna gerarchia, dove cioè tutti i nodi hanno un ruolo identico. Possiamo dunque affermare che le reti peer to peer distribuite sono le reti peer to peer nel vero senso della termine. Il sottogruppo delle reti peer to peer distribuite e che sono anche strutturate, hanno spiccati vantaggi in termini di scalabilità nei confronti del tradizionale modello client-server e delle reti peer to peer distribuite ma non strutturate. Per rete overlay strutturata intendiamo una rete overlay in cui ogni nodo conosce almeno una parte della topologia della rete ma in modo tale che la propagazione dei messaggi avvenga in modo efficiente e quindi scalabile. Diffuse a partire circa dal 2001, le reti overlay distribuite e strutturate consentono di erogare un amplissimo spettro di servizi online a livello globale, dal momento che tali soluzioni peer to peer sono altamente scalabili e lo sono ben di più del modello client-server. Se in quest'ultimo caso il client è solo fruitore e il server è l'erogatore del servizio, nel caso peer to peer ogni utente è sia fruitore che erogatore del servizio. A seconda dell'applicazione che usa il substrato overlay, ogni utente può partecipare nell'erogazione del servizio attraverso il routing di messaggi, storage di contenuti, indicizzazione di contenuti, ecc. Poiché dunque ogni utente partecipa all'erogazione del servizio, questo implica seri problemi di sicurezza. Se una rete è libera di formarsi, poiché ogni host contribuisce a fornire informazioni di routing agli altri host e a instradare i messaggi, è chiaro che un host maligno può modificare a piacimento il proprio contributo all'overlay. Ciò rende le reti overlay distribuite assai

vulnerabili a diversi tipi di attacchi. Essi possono essere suddivisi in tre categorie [1]:

1) Attacco Sybil: una singola entità (l'attaccante) incrementa la propria presenza nel sistema creando nuove identità che poi impiega per operare nella rete.
2) Attacco Eclipse: l'attaccante inquina le informazioni di routing dei nodi onesti in modo che questi inoltrino più messaggi possibili ai nodi controllati dall'attaccante anziché ai nodi corretti.
3) Attacchi di routing e storage: con questa categoria si comprendono vari attacchi a livello applicativo, i quali sono facilitati dai precedenti due attacchi.

E' evidente che l'attacco Sybil è un punto di partenza avvantaggiato per sferrare gli attacchi di tipo 2 e 3.
Approfondiamo ora l'aggettivo "strutturato" delle reti overlay peer to peer strutturate. Il metodo più comune per realizzare una rete overlay distribuita e strutturata è la Distributed Hash Table (DHT). Una DHT è una hash table quindi possiamo idealizzarla come una tabella in cui ogni entry è composta da una chiave (è l'hash) e da un valore. La chiave di una entry di una hash table è l'hash del valore di quella entry. Lo scopo della chiave è quello di identificare univocamente il valore associato, in uno spazio matematico che sia indipendente dalla natura dei valori. In questo modo due valori non trattabili come numeri (ad es. due nomi di file) sono proiettati nell'unico spazio delle immagini della funzione hash usata per calcolare le relative chiavi. I valori di una hash table possono essere un qualunque tipo di dato: contenuti testuali e multimediali, indirizzi IP, username, ecc. Se poi tale stuttura dati è distribuita, significa che è divisa in parti - secondo qualche criterio - e ogni parte è memorizzata in un nodo diverso (host) dell'overlay. Il mapping tra un valore e il nodo che lo memorizza (nodo responsabile) è univocamente determinato dalla chiave di tale valore. A seconda di come è definita la DHT vi sono poi regole protocol-specific che definiscono il mapping chiave-nodo. Vediamo ora che funzionalità offre all'applicazione una DHT. In una DHT vi sono due primitive fondamentali: "put" e "get". Con la prima l'applicazione di un nodo inserisce un nuovo valore nella DHT e il nodo che memorizza tale valore è il nodo responsabile della chiave di tale valore. Con la seconda, l'applicazione di un nodo, data una chiave, recupera il valore associato. Una variante di quest'ultima funzione è la funzione "route" che semplicemente permette ad un messaggio di giungere al nodo responsabile della chiave di destinazione del messaggio. In ogni caso, tutte e tre le operazioni menzionate implicano la propagazione di messaggi. Una DHT pertanto, offrendo le due primitive succitate, può essere il driver ideale per un vasto spettro di applicazioni distribuite. Tali funzionalità però sarebbero poco utili per gestire applicazioni di larga scala se una DHT non

godesse delle seguenti due peculiarità:
1) Scalabilità di memoria: ogni nodo memorizza un'informazione di routing molto piccola rispetto al numero di nodi dell'overlay e tipicamente ha un andamento logaritmico con il numero di nodi.
2) Scalabilità di routing: l'overlay hop count medio dei messaggi è molto basso rispetto al numero di nodi e tipicamente ha un andamento logaritmico con il numero di nodi nel sistema.
Possiamo dunque concludere che la DHT è uno strumento per servire applicazioni distribuite senza la necessità di una struttura gerarchica e in modo altamente scalabile. Le DHT consentono ad esempio il dispiegamento di servizi online serverless su scala globale.
La mancanza di un'entità centrale supervisionante e l'elevata scalabilità di memoria rendono le DHT assai vulnerabili ad attacchi in cui le azioni maligne coinvolgono il routing (ad es. l'attacco Eclipse), in quanto i nodi si affidano a poche informazioni di routing per comunicare con il resto della rete e pertanto è facile per l'attaccante riuscire a popolare l'informazione di routing dei nodi vittima con riferimenti a nodi maligni. La mancanza di un'autorità supervisionante rende anche difficile il controllo degli accessi nel sistema e questo rende ancora più facili i suddetti attacchi sul routing.
Ad oggi come esempi di sistemi reali che impiegano DHT vi sono applicazioni distribuite su scala globale che necessitano la memorizzazione e il recupero di informazioni in modo scalabile. Esempi di queste applicazioni sono alcuni overlay di file sharing (Trackerless BitTorrent, KAD, LimeWire), il sistema VoIP distribuito P2PSIP, diversi motori di ricerca ma anche botnet [1]. I protocolli Chord [2], CAN [3], Pastry [4], Tapestry [5] e Kademlia [6] sono esempi di protocolli overlay che consentono l'instaurazione di una rete peer to peer overlay completamente distribuita e strutturata, in cui la struttura è appunto una DHT.
In [1] vi è una survey sulle principali difese contro gli attacchi comuni alle DHT e si conclude che per rendere sicure le DHT occorre un assegnamenro sicuro degli identificativi ID (chiavi) dei nodi, una bassa percentuale di nodi maligni e sparsi sullo spazio degli ID, replica dei dati e meccanismi di routing resilienti ai nodi dal comportamento scorretto. In ogni caso, il punto più importante è quello di avere un assegnamento degli ID sicuro, al fine di assicurare che la percentuale di nodi maligni sia bassa e che essi non possano scegliere i loro ID in modo da posizionarsi in determinati punti dello spazio degli ID.

Il contributo di questa tesi è quello di risolvere il problema di quello che probabilmente è l'attacco più devastante nelle reti overlay distribuite, ossia l'attacco Eclipse. In questa tesi affrontiamo il problema di identificare e di contrastare l'attacco Eclipse nel caso specifico di rete overlay distribuita strutturata attraverso il protocollo Chord. Quest'ultimo è il protocollo overlay che implementa la DHT che ha finora destato il maggior interesse da parte della

comunità scientifica.

La tesi è costituita dai seguenti capitoli: nel capitolo 2 viene offerta una panoramica della situazione corrente di quanto è stato fatto finora per contrastare l'attacco Eclipse alle reti overlay distribuite. Nel capitolo 3 viene descritto il protocollo Chord, nel capitolo 4 vengono chiariti alcuni termini e notazioni che verrano poi usate nel resto del testo, viene spiegato come abbiamo implemetato l'attacco Elipse in Chord, l'ambiente sperimentale e i valori comuni in tutte le simulazioni. Il capitolo 5 espone la nostra soluzione e i risultati per quanto riguarda l'identificazione dell'attacco Eclipse in Chord. Il capitolo 6 presenta le nostre proposte e i risultati sulle difese contro l'attacco Eclipse al protocollo Chord. Infine nel capitolo 7 vi sono le conclusioni di tutto il lavoro svolto.

2
STATO DELL'ARTE

Una rete overlay è una rete logica sopra una rete fisica in cui i nodi si organizzano creando link virtuali fra loro, quindi, ogni nodo è collegato ad un gruppo di nodi che usualmente chiamiamo vicini (neighbors). Se l'attaccante controlla una frazione sufficiente dei vicini di un nodo onesto, allora può provocare l'attacco "eclisse" (Eclipse attack) nel senso che il nodo vittima avrà una visione ridotta o addirittura nulla degli altri nodi onesti. Questo tipo di attacco è anche noto in letteratura come "routing table poisoning" (avvelenamento della routing table). Un attacco Eclipse può essere usato per lanciare poi attacchi a livello applicativo, che nel contesto delle DHT (Distributed Hash Table) sono gli attacchi di routing e storage, tesi a corrompere l'applicazione distribuita che usufruisce del DHT corrotto.
Sit e Morris [7] sono stati i primi a studiare questo attacco nel contesto delle DHT. Hanno affermato che sistemi in cui non esistono speciali esigenze di controllo sui vicini sono totalmente vulnerabili a questo tipo di attacco. Il modo più facile per sfruttare questa debolezza consiste gli aggiornamenti di routing incorretti. Per esempio, come spiegato in [1], i livelli più alti della routing table del protocollo overlay Pastry necessitano un prefisso comune di poche cifre. Ciò incrementa il numero di identificatori di nodo validi che un attaccante può fornire nei messaggi di routing table update. Questo, al contrario, è più facilmente evitabile in sistemi che impongono forti vincoli di routing table, come ad esempio in Chord, in cui ogni entry della routing table dev'essere il successore di un determinato identificativo nello spazio delle chiavi. Se non vi sono difese opportune, l'attacco Eclipse può far sì che la frazione di entry maligne della routing table dei nodi onesti tenda a uno, dato che il numero di entry maligne può aumentare ad ogni aggiornamento. Altro possibile metodo per realizzare un'eclisse è la sovversione del meccanismo di misura della relazione di vicinanza nella rete. Per esempio in [8] mostrano che un attaccante può ridurre la sua distanza di rete underlay usando un nodo vicino al nodo vittima per inoltrare una "spoofed response" di

heartbeat message. In [9] viene affermato che le misure di prossimità possono anche essere sovvertite mediante vari meccanismi malevoli, ad esempio nel caso in cui l'attaccante controlla una grande infrastruttura di rete come un ISP oppure una grande società. Occorre anche notare che se non è possibile inquinare le misure di prossimità, allora tali relazioni di vicinanza possono essere un aiuto per contrastare l'attacco Eclipse [8]. Altro scenario di attacco di prossimità nelle DHT consiste nel mettere tanti nodi maligni nelle vicinanze dei nodi vittima, in modo che possano infiltrarsi il più possibile nelle routing table di questi ultimi.
Una difesa all'attacco Eclipse si ritiene che abbia pieno successo se la frazione di entry maligne della tabella di routing di un nodo onesto è simile alla frazione f dei nodi maligni presente nel sistema. La ragione di ciò è che f è la frazione di entry maligne che ci si aspetta in un campione di nodi, nell'ipotesi che gli identificatori dei nodi siano generati in modo random uniforme. In ogni caso, l'instradamento di un messaggio usando un singolo path ha difficilmente successo. Per esempio, se f = 0.25 (frazione di nodi maligni nella rete) e la lunghezza del path è 5, la probabilità di avere instradamento con successo è $(1 - 0.25)^5 = 0.24$, che è inaccettabile nella maggioranza delle applicazioni. Perciò, in molte proposte di difesa, i protocolli di aggiornamento della routing table sono accompagnati con alcune forme d'instradamento ridondanti.
Il modello di attacco più usato dalle soluzioni proposte vedono i nodi maligni collidere fra loro e cercano di massimizzare l'inquinamento delle informazioni di routing dei nodi onesti, fornendo sempre informazioni di routing maligne. Comunque, questo non è necessariamente l'unico scenario. Per esempio, l'avversario può provare a attaccare solo un sottoinsieme dei nodi, una chiave particolare, oppure una particolare entry della routing table. L'attaccante può anche solo provare a disseminare moderatamente l'inquinamento, attaccando nodi in modo sequenziale e comportandosi quasi sempre in modo corretto. Nessuna delle soluzioni proposte sono state pensate contro questi comportamenti insidiosi. In [9] gli autori discutono gli attacchi localizzati, i quali richiedono che un nodo onesto sia circondato dai nodi maligni in termini di distanza di rete. Essi concludono che difendersi contro tali attacchi rimane un problema aperto. Dalla revisione della letteratura della sicurezza in reti overlay peer to peer, emerge che la migliore difesa base all'attacco Eclipse consiste nel vincolare gli identificatori dei nodi che possono essere usati nella tabella di routing come è già naturale in Chord. Questo è valido solo se gli identificatori dei nodi sono random e stabili (cioè in una rete non dinamica), e i nodi maligni sono uniformemente sparsi nello spazio degli identificatori. Per raggiungere questi condizioni, l'approccio più semplice è l'impiego di nodi con identificatori stabili distribuiti da un'autorità fidata. In alternativa, in [11] propongono un approccio basato sul churn, causando ogni volta che un nodo entra nella rete il riassegnamento degli identificatori dei nodi già presenti. Il prezzo da pagare è che sfruttando le proprietà degli

identificatori dei nodi come unico criterio per selezionare le entry da inserire nella routing table, si impedisce l'ottimizzazione delle prestazioni come la selezione dei vicini in termini di rete underlay. L'ottimizzazione di network proximity può essere facilmente implementata in sistemi come Pastry perché impongono deboli requisiti di geometria overlay alle entry della routing table. La conseguenza è che tanti nodi soddisferanno i requisiti degli identificatori (geometria overlay), e dunque le misure di rete possono essere utili per selezionare i migliori vicini underlay. Il problema di ciò è che i nodi maligni possono facilmente popolare le routing table dei nodi onesti, per mezzo di azioni maligne sui protocolli di routing table update oppure sul sistema di misura di prossimità. In [12] viene mostrato che avendo 15% di nodi maligni nella rete overlay corrisponde ad avere approssimativamente 80% di entry maligne nelle tabelle di routing del protocollo Pastry. La maggiore parte della letteratura sull'attacco Eclipse è mirata a difese che tentano di preservare le prestazioni della rete su routing table basate su quelle di Pastry. In questi casi, l'approccio più comune è l'uso ridondante degli entry della routing table [8], in cui è ci si aspetta che alcune entry saranno oneste e sufficienti per permettere un routing ridondante con successo. In [12] invece gli autori propongono di costringere periodicamente ogni nodo di uscire dalla rete overlay e a rientrare con un nuovo identificatore, e con due nuove tabelle di routing, una delle quali sarà ottimizzata, finché il nodo non esegue la nuova join prevista. Questo è simile all'approccio di [11]. Altra soluzione per contrastare l'attacco Eclipse è quello di controllare il numero di link entranti (grado di ingresso) e uscenti per nodo (grado di uscita). Il ragionamento è che il grado di ingresso dei nodi maligni è in media superiore al grado dei nodi onesti poiché questi ultimi si ritrovano, inconsapevolmente, a puntare a molti nodi collusi anziché a nodi non collusi. L'approccio secondo [10] ha il vantaggio di essere completamente decentralizzato, ma impone un basso limite di grado di uscita, il quale corrisponde ad un aumento del tempo di lookup in assenza di attacco. Si evince quindi, che difendersi dall'attacco Eclipse implica un trade-off tra prestazioni e complessità. In [9] vengono affrontati diversi problemi sulla sicurezza di reti P2P strutturate, quale è il protocollo Chord, e nel nostro ambito di attacco Eclipse, ci interessano le soluzioni di mantenimento sicuro delle informazioni di routing e l'inoltro sicuro dei messaggi. Per il primo aspetto si osserva che, se è un protocollo prevede una routing table che rispetti determinate regole di geometria overlay, allora il nodo rifiuta le informazioni di routing update che sembrano non rispettare tali regole. Poiché questo mantenimento sicuro della routing table è verosimilmente non perfetto - ci sono ancora entry maligne – viene allora proposta una possibile soluzione per l'inoltro sicuro dei messaggi. La difesa proposta è di tipo reattivo: prima c'è un test e poi, se necessaria, la reazione. Il test è un test a soglia e consiste nel misurare quanto il nodo che ha ricevuto un messaggio è verosimilmente il vero destinatario. La reazione è un routing ridondante. Queste soluzioni in

[9] vengono descritte per il solo caso del protocollo Pastry ma sono estensibili ad altri protocolli come ad esempio Chord. Il test consiste nel misurare la densità di nodi (nello spazio delle chiavi) di un certo numero di nodi attorno al nodo destinazione per compararla con la densità attorno al nodo sorgente. Poiché quella dei nodi maligni è inferiore di quella dei nodi onesti, il nodo sorgente può dedurre da questo test se il messaggio è finito su nodi maligni o no. Questo meccanismo è simile a quanto noi facciamo in Chord: la densità del nodo sorgente la stimiamo come media delle distanze fra le entry della successor list, mentre come densità al nodo destinazione prendiamo il campione dato dalla distanza fra la chiave di destinazione del messaggio e il nodo su cui è terminato. Non si può fare altrimenti, infatti il nodo sorgente non può fare stime di densità della destinazione in quanto non ne conosce il neighborhood. E anche se glielo chiedesse con messaggi appositi, un nodo maligno può sempre fornire informazioni ingannevoli. La reazione presentata in [9], invece, consiste nel mandare copie dello stesso messaggio finché non si raggiunge un nodo avente la destinazione nel proprio neighborhood. A quel punto si sfrutta l'informazione dettagliata che tale nodo ha sulla regione dello spazio delle chiavi attorno alla destinazione (root) per assicurare che tutte le destinazioni replica (replica roots) ricevano un copia del messaggio. Sarebbe interessante vedere i risultati in Chord, tuttavia siamo di fronte ad un sistema ridondante, il quale raggiunge sì ottimi risultati (almeno in Pastry) ma con un overhead potenzialmente elevato. Inoltre l'attacco in esame su Pastry è anche assai meno devastante dell'attacco Eclipse in Chord, in quanto la probabilità di routing corretto è ben inferiore nell'ultimo caso a parità di percentuale di nodi maligni, come dimostrato dai grafici. Infine, come avevamo accennato, sappiamo che la difesa ideale all'attacco Eclipse garantisce che la probabilità che un entry della tabella sia maligna è equivalente alla frazione f di nodi maligni nella rete overlay. Questo significa che le tecniche fino ad oggi proposte non sono sufficienti a garantire il funzionamento appropriato delle DHTs in caso di attacco Eclipse a meno che non siano combinate con altri meccanismi come il routing ridondante. Finora abbiamo fatto una rapida overview di come è stato affrontato il problema dell'attacco Eclipse nelle reti overlay peer to peer in generale. Nel caso specifico del protocollo Chord vi è ad oggi sorprendentemente poco in letteratura. In [13] viene presentata una soluzione per aumentare la probabilità di successo delle query Chord in presenza di nodi che semplicemente eliminano ogni messaggio dati che ricevono. Questo non è propriamente un attacco Eclipse poiché in quest'ultimo l'azione maligna è il routing modificato dei maligni su tutti i messaggi (controllo e dati) e non la semplice eliminazione dei messaggi dati. Come dimostrano anche i risultati numerici dell'attacco di data dropping, l'attacco Eclipse è ben peggiore, quindi il contesto di partenza di [13] è molto più favorevole rispetto al nostro caso. La soluzione proposta è quella di memorizzare presso ogni nodo dei path ciclici appresi

dall'osservazione del path che viene appositamente inserito nei messaggi. Un ciclo è infatti esente dai maligni per il semplice fatto che è un path che torna al nodo in questione, quindi non può aver incontrato nessun maligno. Di conseguenza, ad ogni lookup, prima di invocare il Chord routing, si sceglie come next hop il nodo che fra tutti i cicli conosciuti dal nodo precede più da vicino la chiave di destinazione del messaggio e che allo stesso tempo è presente nella finger table o nella successor list. Se una tale nodo non viene trovato, il messaggio è inoltrato alla finger o al successor scelti in base al normale Chord lookup. Questa potrebbe essere una soluzione interessante anche contro l'attacco Eclipse, tuttavia, come già detto, è stata sperimentata sull'attacco di data dropping anziché sull’attacco Eclipse.

3
IL PROTOCOLLO CHORD

3.1 Introduzione

I sistemi e le applicazioni peer to peer sono sistemi distribuiti senza alcun controllo centralizzato o organizzazione gerarchica, dove l'esecuzione del software in ogni nodo è equivalente in funzionalità. Una panoramica sulle recenti applicazioni peer to peer dà luogo ad una lunga lista: memorizzazione ridondante, selezione dei server vicini, anonimità, ricerca, autenticazione, naming gerarchico, ecc. Nonostante questo variegato panorama, l'operazione centrale nella maggior parte dei sistemi distribuiti è la localizzazione efficiente dei dati. Il protocollo Chord è un protocollo scalabile per la ricerca in sistemi dinamici peer to peer con frequenti arrivi e uscite di nodi. Chord supporta un'operazione fondamentale: data una chiave, assegna tale chiave ad un nodo. In base all'applicazione che usa Chord, il nodo potrebbe essere responsabile per memorizzare un valore associato alla chiave. Chord usa una variante del "consistent hashing" per assegnare le chiavi ai nodi. Il "consistent hashing" tende a bilanciare il carico, dato che ogni nodo riceve all'incirca lo stesso numero di chiavi, e causa un relativamente piccolo movimento di chiavi quando i nodi entrano o escono dal sistema. I lavori precedenti sul "consistent hashing" assumono che i nodi siano informati sulla maggior parte dei nodi del sistema, rendendo impraticabile la scalabilità con un elevato numero di nodi. All'opposto, ogni nodo Chord ha bisogno di informazioni di "routing" riguardo ad un piccolo numero di nodi. Tre caratteristiche che distinguono Chord da altri protocolli di ricerca peer to peer sono la semplicità, la correttezza e le performance.

3.2 Descrizione del protocollo

Il protocollo Chord specifica come trovare le locazioni delle chiavi, come i nodi entrano ed escono dal sistema e come comportarsi in caso di uscita non intenzionale. Innanzitutto Chord fornisce una computazione veloce distribuita di una “hashing function” che assegna le chiavi ai nodi responsabili per esse. Usa “consistent hashing” che gode di diverse buone proprietà. Con alta probabilità la funzione hash bilancia il carico (tutti i nodi ricevono all’incirca lo stesso numero di chiavi). Altrettanto con alta probabilità, quando un N-esimo nodo entra (o esce) dalla rete, soltanto una frazione O(1/N) di chiavi viene trasferita in un altro nodo, il che è chiaramente il minimo necessario per mantenere un carico bilanciato. Chord fornisce la scalabilità del “consistent hashing” evitando il requisito che ogni nodo conosca tutti gli altri nodi. Un nodo Chord ha necessità di un piccolo numero di informazioni di “routing” relativamente agli altri nodi. Dal momento che questa informazione è distribuita, un nodo risolve la funzione hash comunicando con un piccolo numero di nodi. In una rete di N-nodi, ogni nodo mantiene informazioni su circa $O(\log_2 N)$ nodi, e la ricerca richiede al massimo $O(\log_2 N)$ messaggi.

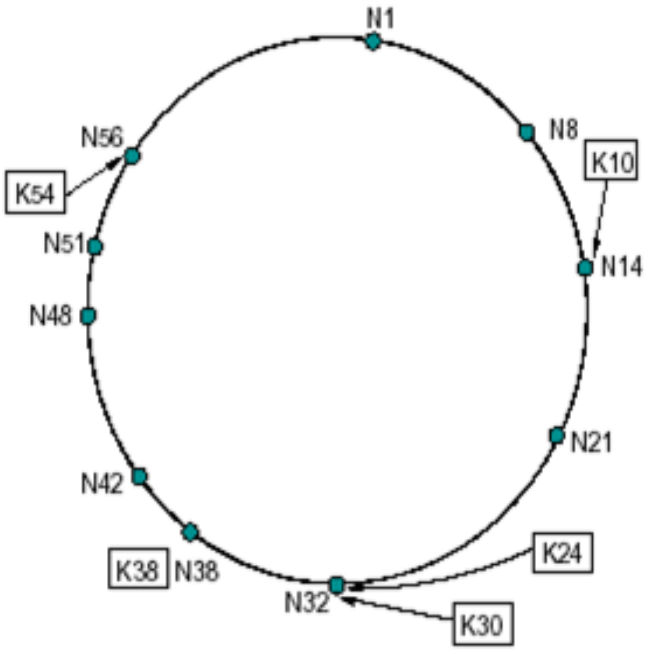

Fig.1. *Anello delle chiavi in cui vi sono 10 nodi e 5 chiavi.*

Consistent hashing

La funzione di “consistent hashing” assegna ad ogni nodo e ad ogni chiave un identificativo di m bit usando una funzione di hash come SHA-1 (è una funzione univoca che stabilisce la

corrispondenza tra volore e chiave). L'identificativo di un nodo è scelto calcolando l'hash dell'indirizzo IP del nodo, mentre l'identificativo di una chiave è prodotto calcolando quello della chiave. Assumiamo per comodità il termine "chiave" per indicare la chiave e la sua immagine dovuta alla funzione hash, vale la stessa considerazione per il "nodo". La lunghezza m dell'identificativo deve essere abbastanza perché la probabilità che due nodi o chiavi abbiano lo stesso hash sia molto bassa. L'hashing consistente assegna le chiavi ai nodi come segue(la regola di chord), Gli identificativi sono ordinati in un "identifier circle" modulo 2^m. La chiave k è assegnata al primo nodo il cui identificativo è uguale o segue k nello spazio degli identificativi. Questo nodo è chiamato "successor node" della chiave k, denotato da successor(k). Se gli identificativi sono rappresentati come un cerchio di numeri da 0 a 2^{m-1}, allora successor(k) è il primo nodo in senso orario da k. Il circle ha 10 nodi collegati alla rete (quelli con i puntini neri). Attraverso la Fig.1 si comprende come avviene l'assegnamento delle risorse. Il successore dell'identificativo 1 è 1, così la chiave 1 dovrebbe essere allocata al nodo 1. Similmente la chiave 10 dovrebbe essere allocata al nodo 14, e la chiave 54 al nodo 56.

Simple key localization

Un numero veramente piccolo di informazioni di routing è necessario per implementare "consistent hashing" in un ambiente distribuito. Ogni nodo ha necessità di conoscere solo il nodo successore nel "identifier circle". Le richieste per un dato id possono essere passate lungo l'anello attraverso i puntatori ai successori fino a che non incontrano un nodo che succede l'id. Una porzione del protocollo Chord mantiene questi puntatori ai successori, in modo da far sì che ogni ricerca sia risolta correttamente. Tuttavia questo è un approccio non ottimale: potrebbe richiedere di attraversare tutti gli N nodi per trovare l'assegnazione adeguata. Come indica la fig.3, per consegnare un messaggio alla sua destinazione, è sufficiente che ogni nodo conosca solo il nodo che lo segue nell'anello.

```
// ask node n to find the successor of id
n.find successor(id)
if (id ∈ (n, n.successor])
    return n.successor;
else
    // forward the query around the circle
    return successor.find_successor(id);
```

Fig. 2: *Algoritmo di simple node localization.*

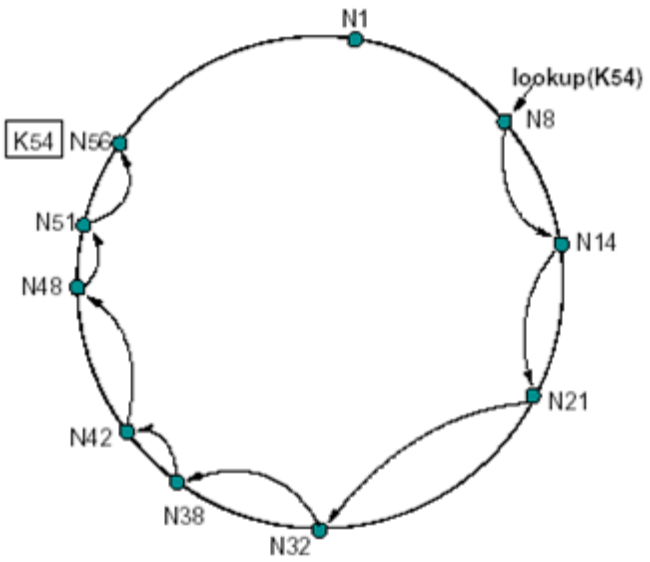

Fig.3: *Esempio in cui il nodo di chiave 8 cerca il nodo responsabile della chiave 54, secondo la localizzazione semplice delle chiavi.*

Scalable key localization

Per accelerare questo processo, Chord mantiene un'informazione di routing addizionale. Questa informazione addizionale non è essenziale per la correttezza, che è garantita nel momento in cui l'informazione sul successore viene mantenuta correttamente. Come prima, consideriamo m il numero di bit nell'id nodo/chiave. Ogni nodo, n, mantiene una tabella di routing con al più m elementi, chiamata "finger table". L'i-esimo elemento nella tabella del nodo n contiene l'identità del primo nodo, s, che succede n di almeno 2^{i-1} nel "identifier circle", ovvero s = successor(n + 2^{i-1}) dove 1 ≤ i ≤ m (tutto moldulo 2^m). Chiamiamo il nodo s iesimo finger del nodo n. Un elemento della finger table include insieme il Chord identifier e l'indirizzo IP (e numero di porta) del nodo rilevante. Notare che il primo finger di n è l'immediato successore nel "identifier circle"; per convenienza ci riferiremo ad esso come al successore piuttosto che al primo finger. Cosa succede se un nodo cerca una chiave k di cui non è il responsabile? Il nodo cerca nella sua finger table il nodo j che precede immediatamente k e gli inoltra una richiesta per successor(k).

```
// ask node n to find the successor of id

n.find_ successor(id)
if (key ∈ (n, n.successor])
      return n.successor;
else
      n' = closest _preceding _node(id);
      return n'.find_successor(id);

// search the local table for the highest predecessor of id

n.closest_ preceding node(id)
    for i = m downto 1
       if (finger[i] ∈ (n, id))
          return finger[i];
    return n;
```

Fig. 4: *Algoritmo di scalable node localization.*

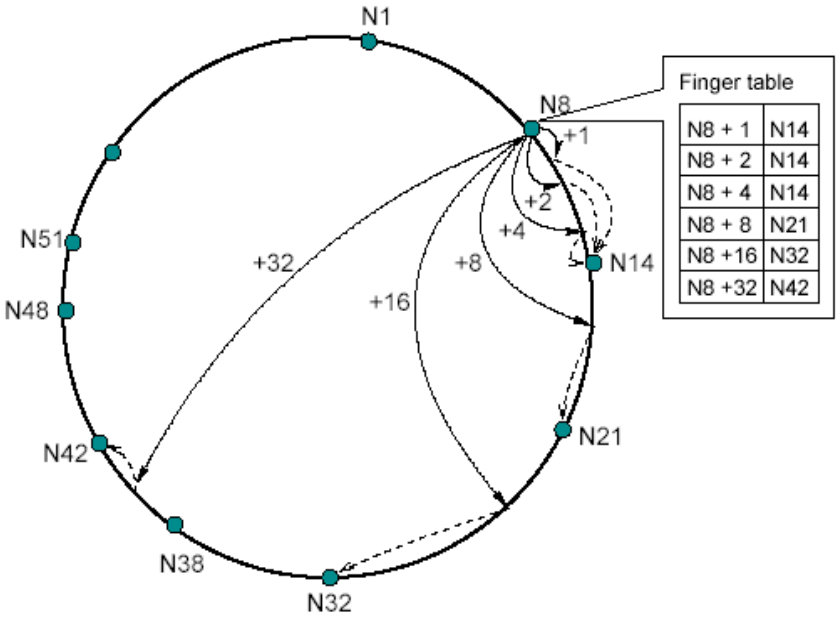

Fig. 5. *Finger table del nodo di chiave 8.*

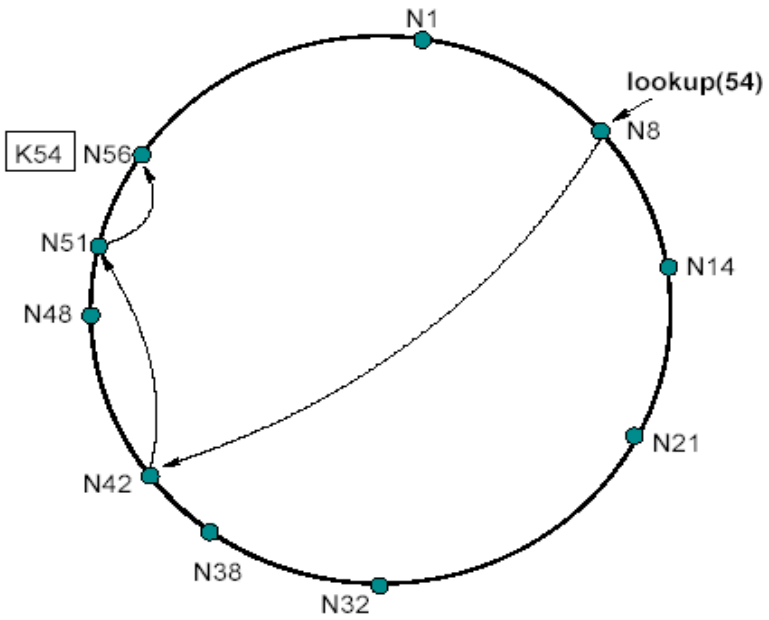

Fig. 6: *Percorso dello stesso messaggio di fig. 3 ma questa volta mediante localizzazione scalabile delle chiavi.*

Nell'esempio della fig. 5, possiamo vedere tutte le informazioni di routing contenute nella finger table del nodo N8. Supponiamo che il nodo N8 voglia trovare il successore della key (54). Visto che nella routing table del nodo N8, il closest preceding node è il nodo N42. Il nodo N8 chiederà al nodo N42 di risolvere la query find-successor(54), il nodo N48 al suo suo turno controllerà nella sua finger table e chiederà il closest preceding node al successor della key (54) di risolvere la query. Finalmente il nodo N51 restituirà al nodo N8, il nodo N56 che è il successore della key (54).

Vi è da specificare infine che, in un generico overlay, esistono 3 modalità di routing:

1) Iterativa: l'avanzamento del messaggio (o query) da un hop al successivo hop avviene passando per il nodo sorgente. In altre parole, il nodo hop i non inoltra direttamente il messaggio all'hop successivo i + 1, bensì comunica al nodo sorgente tale hop successivo. E' poi il nodo sorgente che invia il messaggio all'hop successivo e così via sino a quando giunge a destinazione.

2) Semi – ricorsiva: i nodi intermedi del path inoltrano direttamente il messaggio senza coinvolgere il nodo sorgente (fig. 6). La risposta, se prevista, va direttamente dal nodo destinazione al nodo sorgente.

3) Ricorsiva: è come la semi ricorsiva con la differenza che la risposta non va direttamente dalla destinazione al sorgente, bensì è instradata in modo ricorsivo finché non arriva al sorgente.

Join and Leave

Le due operazioni fondamentali che interessano un qualsiasi nodo in Chord sono la *join* (ingresso nella rete) e la leave (uscita dalla rete); tali operazioni, proprio per la dinamicità di Chord, potranno essere effettuate da ciascun nodo in ogni momento. L'obiettivo principale, nel momento in cui un nodo entra o esce dalla rete, è quello di evitare di spostare un elevato numero di chiavi mantenendo però ordinato il mapping delle chiavi sui nodi per garantire di rendere possibile il lookup delle risorse. In modo da rendere la ricerca più veloce, è anche desiderabile che le finger table siano tenute corrette. Questa sezione mostra come mantenere questi due invarianti quando un singolo nodo entra nel sistema. Prima di descrivere le operazioni di entrata, mostriamo le sue performance. Per semplificare i meccanismi di entrata e uscita, ogni nodo in Chord mantiene un puntatore al predecessore. Il puntatore al predecessore di un nodo contiene il Chord identifier e l'indirizzo IP dell'immediato predecessore di tal nodo, e può essere usato per percorrere in senso antiorario l'"identifier circle". Per fare questo, Chord deve garantire che per semplificare le operazioni ogni nodo tiene conto del suo predecessore, le operazioni da svolgere al join di un nodo n sono:

- Inizializzare predecessore e finger di n
- Aggiornare le finger e i predecessori degli altri nodi per tenere conto di n (cioè il nodo n deve entrare nelle finger table degli altri)
- Trasferimento dei dati (cioè muovere sul nodo n tutte le chiavi di cui è il successore, n diventa responsabile solo delle chiavi contenute nel suo succesore)

Ipotizziamo che il nuovo nodo impari l'identità di un nodo Chord esistente n' da qualche meccanismo esterno. Il nodo n usa n' per inizializzare il suo stato e aggiungere se stesso alla rete Chord esistente, come segue.

Inizializzazione predecessor e fingers

Ricavare la finger table è un calcolo facile da eseguire. Abbiamo già scritto in precedenza che, per determinare le varie entry r il nodo n, sarà necessario applicare sempre la stessa formula $n + 2^i$ con $1 \leq i \leq m$, $m = \log_2 N$ dove N è il numero dei nodi che costituiscono l'anello e m il numero di bit dell'indentifier della chiave. Per calcolare il predecessore invece dovranno essere fatte tre operazioni:

1) *n* chiede ad un nodo qualsiasi di calcolare n'=*successor* (n);
2) *n* chiede a n' chi è il suo predecessore
3) Il predecessore di *n'* diventa ora predecessore di n

Per ottimizzare il processo, il nodo appena entrato può chiedere ad un suo vicino il suo nodo precedente e una copia della finger: usando tali dati può ricostruire la propria finger dal momento che quella del nodo vicino sarà molto simile, riducendo così i tempi per la costruzione della nuova finger dell'ordine di $O(\log_2 N)$.

Aggiornamento predecessori e fingers degli altri nodi nella rete

La funzione update_finger_table aggiorna la finger table. Supponiamo che, dopo l'ingresso di n nella rete, esso debba diventare l'i-esima finger di un nodo *p*, e questo può accadere se e solo se: p precede n di almeno 2^{i-1}, l'i-esima finger del nodo p succede n. Il primo nodo p, che unisce queste due condizioni è l'immediato predecessore di $n - 2^{i-1}$. Così, per un dato n, l'algoritmo inizia con l'i-esima finger del nodo n e continua a camminare in senso anti orario sul "identifier circle" finchè non incontra un nodo il cui i-esimo finger precede n.

Trasferimento delle chiavi

Questa operazione può essere portata a termine con una semplice ispezione di tutte le chiavi gestite da successor(n); L'ultima operazione che deve essere eseguita quando un nodo entra nella rete è spostare tutta la responsabilità sulle chiavi di cui ora il nuovo nodo è il successore. Cosa esattamente questo comporti dipende dal software di più alto livello usando Chord, ma tipicamente potrebbe riguardare lo spostamento dei dati associati ad ogni chiave nel nuovo nodo. Il nodo n può diventare il successore solo delle chiavi per cui era precedentemente responsabile il nodo che segue n immediatamente, così solo n deve contattare quel nodo per trasferire la responsabilità per tutte le chiavi rilevanti. La fig. 7 è un esempio dell'operazione di join del nodo N26 nel sistema e le operazioni che seguono la sua entrata.

- Il nodo N26 entra nel sistema
- Il nodo N26 sceglie il nodo N32 come il suo successore
- Il nodo N26 contatta il nodo N32
- Il nodo N32 sceglie il nodo N26 come il suo predecessore
- Il nodo N26 coppia dal nodo N32 tutte le chiavi di cui è responsabile

- Il nodo N21 esegue la funzione di stabilize() e chiede al suo successore chi è il suo predecessore, il quale è il nodo N26.
- Il nodo N21 sceglie il nodo N26 come successore
- Il nodo N21 informa il nodo N26 della sua presenza
- Il nodo N26 sceglie il nodo N21 come predecessore

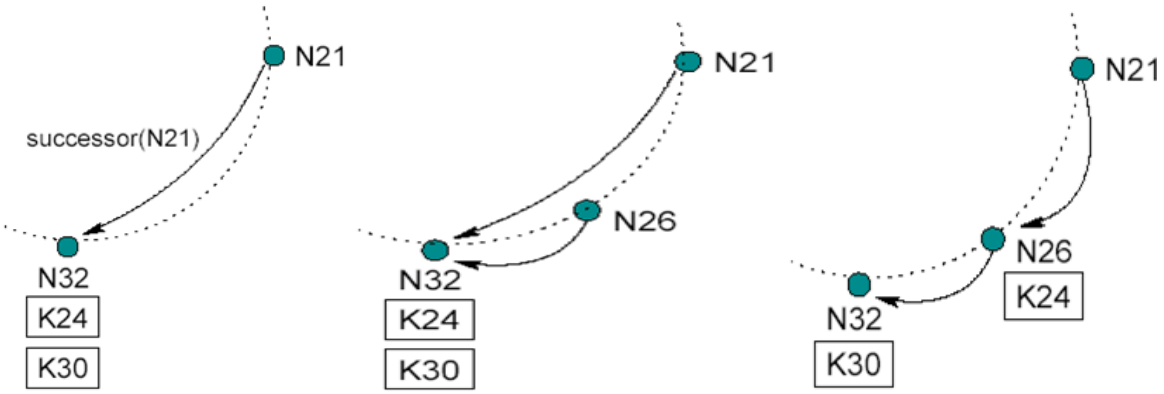

Fig.7. *Esempio che illustra la join del nodo di chiave 26.*

Il protocollo di stabilizzazione è costituita da 6 funzioni che sono:

- Create()
- Join()
- Stabilize()
- Notify()
- Fix_fingers()
- Check_predecessor()

Si riportano, di seguito, in pseudocodice le operazioni che interessano la join e la stabilizzazione di un nodo:

Create(): crea una nuova istanza di anello che andrà a contenere i vari nodi della rete.

```
n.create()
predecessor = nil;
successor = n;
```

Fig. 8: *Pseudocodice della funzione create() eseguita da un nodo n.*

Join(): permette ad un nodo di accedere ad una rete già esistente. Un nodo appena entrato nella rete chiede ad un qualsiasi nodo m di trovare il suo diretto successore.

```
n.join(n')
  predecessor = nil;
  s = n'.find_successor(n);
  build_fingers(s);
  successor = s;
```

Fig. 9: *Pseudocodice della funzione join() eseguita da un nodo n.*

Stabilize(): funzione chiamata periodicamente per aggiornare lo stato della rete e per notificare l'aggiunta di nuovi nodi. A tal fine viene determinato il successore del predecessore di n e se si sono aggiunti nuovi nodi viene fatta la notifica.

```
// periodically verify n's immediate successor,
// and tell the successor about n.

n.stabilize()
x = successor.predecessor;
if (x ∈ (n,successor))
    successor = x;
successor.notify(n);
```

Fig. 10: *Pseudocodice della funzione stabilize() eseguita da un nodo n.*

Notify(): al nodo n viene settato come predecessore il nodo m.

```
// n' thinks it might be our predecessor.
n.notify(n')
if (predecessor is nil or n' ∈ (predecessor;n))
      predecessor = n';
```

Fig. 11: *Pseudocodice della funzione notify() eseguita da un nodo n.*

Fix_finger(): periodicamente viene chiamato questa funzione per aggiornare le entries della finger table.

```
n.fix_fingers()
  next = next + 1;
  if (next > m)
     next = 1;
  finger[next] = find_successor(n + 2^(next-1));
```

Fig. 12: *Pseudocodice della funzione fix_fingers() eseguita da un nodo n.*

Check_predecessor(): periodicamente è controllata la presenza del predecessore.
Consente ad *n* di aggiornare il proprio predecessor, nel caso che questo sia fallito. In caso di fallimento, il predecessor viene settato quando *n* riceve una *notify* (durante una procedura *stabilize*).

```
n.check_predecessor()
   if (predecessor has failed)
      predecessor = null;
```

Fig. 13: *Pseudocodice della funzione check_predecessor() eseguita da un nodo n.*

Chord leave

Siccome il protocollo descritto finora è corretto anche in caso di fallimenti per i nodi potremmo gestire la leave come un fallimento, tuttavia è preferibile, per migliorare le prestazioni del sistema, adottare alcuni accorgimenti, quando un nodo si disconnette volontariamente dalla rete – Il nodo che lascia la rete trasferisce le proprie risorse al suo successore, notifica ai propri vicini (predecessore e successore) che sta uscendo dal sistema. In particolare comunica il predecessore al suo successore e il successore al suo predecessore, in questo modo i due vicini possono connettersi senza utilizzare la stabilize/notify. L'uscita di un nodo n per un qualche motivo fallisce e quindi deve lasciare la rete, tutti gli altri nodi che hanno nelle loro fingers il riferimento al nodo *n*, dovranno modificare le entries della finger e dovranno andare a determinare il successore di *n*. Inoltre,

il fallimento di un nodo non deve andare a perturbare le query che sono in corso in quel momento: per questo motivo quindi il sistema si deve subito ristabilizzare. A tal fine, ogni nodo contiene oltre alla finger table una lista dei successori e, qualora il diretto successore dovesse per un qualche motivo abbandonare per fallimento l'anello, mediante la lista dei successori potrà aggiornare e modificare il nodo diretto successore. Potrebbe anche succedere che nell'intervallo di tempo tra il fallimento di un nodo e la completa stabilizzazione della rete, un qualche altro nodo potrebbe inviare richieste al nodo caduto o servirsi del nodo caduto per effettuare lookup su un altro nodo: per evitare ciò, idealmente la richiesta dovrebbe essere processata, dopo un certo timeout, passando per un altro percorso che non è interessato dal fallimento. Ciò in molti casi avviene grazie ad un'ulteriore lista di nodi facilmente costruibile a partire dalla finger del nodo precedente decaduto. Se si usa una lista di successori di lunghezza $r = O(\log_2 N)$ in una rete inizialmente stabile e in cui successivamente un nodo fallisce con probabilità 0.5, allora con alta probabilità find_successor(k) sarà allora il nodo ancora attivo più vicino a k.

4

ATTACCO ECLIPSE AL PROTOCOLLO CHORD

4.1 Termini e notazioni

Di seguito elenchiamo delle assunzioni che, se non diversamente specificato, sono sempre valide in tutta le presente tesi.

1) Il neighborhood di un nodo n in Chord è l'unione della finger table, della successor list e del predecessor, ovvero l'insieme di tutti i nodi di cui il nodo n assume l'esistenza. Ogni elemento n' del neighborhood del nodo n è rappresentato dal socket con cui il nodo n comunica direttamente con il Chord layer del nodo n'.
2) Il routing scelto è di tipo semi-recursive. Si veda il capitolo su Chord per la descrizione delle diverse modalità di overlay key-based routing.
3) La lunghezza delle chiavi è m bit. Il paper di Chord [2] propone di ottenere le chiavi come hash SHA-1 quindi m = 160 bit dell'indirizzo IP del nodo. Nelle nostre simulazioni abbiamo m = 32 bit.
4) Il numero di nodi nell'overlay è N, definito dunque come N = numero nodi maligni + numero nodi onesti.
5) f = frazione di nodi maligni (numero nodi maligni / numero nodi maligni + numero nodi onesti).
6) Un messaggio si dice ricevuto da un nodo n quando arriva al nodo n e tale messaggio può essere sia transitante che terminante sul nodo n.

7) In Chord con il termine “risposta” intendiamo un messaggio che un nodo d su cui termina un messaggio restituisce al nodo sorgente di tale messaggio. L’unico messaggio definito in Chord avente tale caratteristica è la fix finger response, con la quale un nodo aggiorna una finger della propria finger table. Noi per risposta, oltre alla fix finger response, ci riferiamo anche ai messaggi di acknowledgment (overlay ack) dei messaggi applicativi quando terminano sul loro nodo destinazione. In Chord non sono definiti tali overlay ack, tuttavia non vi è nulla in contrario a fare l’ipotesi della loro presenza. Non dimentichiamo che Chord nasce come protocollo per rispondere a query su sistemi distribuiti, e una query implica infatti una risposta. Nel nostro caso tuttavia, per overlay ack non ci riferiamo propriamente a messaggi dell’applicazione che impiega Chord, intendiamo piuttosto messaggi di livello overlay (Chord) in quanto essi ci occorrono per ottenere specifiche funzioni di identificazione e difesa da parte del protocollo Chord stesso.
8) Per nodi maligni intendiamo i nodi collusi fra loro, ossia nodi che evadendo dalle regole di routing del protocollo, vedono la rete come se fosse popolata unicamente dai nodi a cui collidono. Essi rappresentano il subset di nodi di proprietà o che sono controllati dall’attaccante.

4.2 Implementazione dell’attacco

4.2.1 Premessa sull'attacco Eclipse

Con il termine rete, se non diversamente specificato, intendiamo sempre rete overlay Chord ovvero una rete in cui nodi generano, ricevono e instradano messaggi overlay cioè messaggi che utilizzano indirizzi overlay (le chiavi).
Ogni protocollo di rete prevede uno scambio di diversi tipi di messaggi di controllo, ovvero di messaggi che non trasportano payload applicativo ma semplicemente informazioni di controllo dello stesso layer protocollare. Le funzioni di controllo del protocollo Chord (join, fix_finger, stabilize, notify) comportano lo scambio di informazioni a livello overlay. Nel caso particolare di Chord tali informazioni sono sempre i socket dei nodi. Dall’indirizzo IP del socket si deriva univocamente la chiave del nodo. Nell'ambito di network security queste informazioni (chiavi) si dividono in due categorie:

1) Informazioni vere
2) Informazioni false

Le informazioni vere sono il risultato dell'esecuzione del protocollo in due casi:

1.1) Il protocollo Chord originale, quindi onesto. Tutte le informazioni generate da un nodo onesto sono sempre vere finché tale nodo, al fine di generare tale risposta, si avvale esclusivamente di informazioni vere.
1.2) Il protocollo Chord modificato, quindi maligno. Un nodo maligno cerca il più possibile di dare risposte vere fintantoché sono maligne. La ragione è che l'attacco è implementato in modo da aderire il più possibile al protocollo originale, così da massimizzare la difficoltà di riconoscimento e di difesa dell'attacco.

Un nodo onesto ha informazioni false se ha ricevuto almeno una risposta falsa da un nodo onesto o maligno. Le informazioni false sono il risultato dell'esecuzione del protocollo in due casi:
2.1) Il protocollo Chord originale, quindi onesto. Se una risposta di un nodo onesto ha comportato l'utilizzo di informazioni false a disposizione da tale nodo, allora la sua risposta sarà falsa. Una risposta falsa è necessariamente maligna poiché l'origine della propagazione delle informazioni false è sempre un nodo maligno, a meno di errori involontari negli onesti.
2.1) Il protocollo Chord modificato, quindi maligno. Se la call di un nodo onesto comporta una response onesta di un maligno secondo il protocollo originale, allora il nodo maligno devia dal protocollo originale allo scopo di fornire un informazione maligna e tale informazione è falsa dato che non è conforme alle regole del protocollo.

Un informazione maligna può essere sia vera (corretta) che falsa (non corretta) ma un nodo onesto che riceve un informazione maligna non sa se è vera o falsa. Se così fosse potrebbe accettare tutte le informazioni oneste e rifiutare tutte le informazioni maligne. Esempio: un maligno risponde ad una richiesta di costruzione di una finger (fix finger). Se il vero successor s del valore start della fix finger response è un nodo maligno allora il maligno risponde con l’informazione vera, se invece s è onesto il maligno non risponde con s ma con il successor maligno del valore start che non è il vero successor di start, perciò l’informazione fornita dal maligno è falsa. Un altro concetto essenziale nell'attacco Eclipse è l'inquinamento di un nodo. L'inquinamento di un nodo è la percentuale maligna di informazione nelle strutture dati usate per l'instradamento (overlay routing poisoning). Tanto più un nodo onesto è inquinato e tanto più devia il traffico verso i maligni. L'inquinamento di un nodo ha origine nelle informazioni di controllo maligne che il nodo riceve, non importa se vere o false.

4.2.2 Dall'attacco Sybil all'attacco Eclipse

L'attacco Eclipse è un'evoluzione dell'attacco Sybil. In quest'ultimo l'attaccante assume diverse identità, vale a dire che è presente nella rete con una moltitudine di nodi anziché uno solo. L'attaccante, assumendo diverse identità nella rete, non è in prima battuta Naturalmente, all'aumentare della quantità di nodi con cui l'attaccante è presente nella rete, aumenta il suo controllo della rete. Nell'attacco Sybil non vi è nessun comportamento maligno nei nodi dell'attaccante, infatti i nodi dell'attaccante eseguono il protocollo originale e si comportano esattamente come gli onesti. Nell'attacco Sybil non vi è nessun tipo di intelligenza se non nel fatto che l'attaccante è in grado di essere presente con più nodi nella rete anziché uno solo. Le chiavi catturate dall'attaccante sono dunque soltanto le chiavi di cui i nodi posseduti dall'attaccante sono i destinatari da protocollo. In altre parole, i nodi dell'attaccante non catturano chiavi a loro non destinate. E' esattamente e solo questa la differenza con l'attacco Eclipse: nell'attacco Eclipse i nodi maligni catturano tutte le chiavi che ricevono, anche quelle a loro non destinate mentre nell'attacco Sybil l'attaccante instrada normalmente le chiavi ad esso non destinate. La conseguenza è che sia l'inquinamento a livello overlay sia il comportamento malevolo sui messaggi applicativi è assai più presente nell'attacco Eclipse che nel Sybil. Il grado di efficacia dell'attacco Sybil dipende unicamente dalla percentuale di nodi maligni, cioè i nodi in possesso dell'attaccante. La percentuale di chiavi catturate dai maligni nel caso di attacco Sybil cresce linearmente. Il motivo è semplice: un nodo maligno n cattura solo la percentuale di chiavi di cui esso è successore ovvero se n è la chiave del nodo n, allora le chiavi di cui è successore sono quelle nell'intervallo (p,n] dove p è il predecessor di n. Per il consistent hashing, le chiavi dei nodi cadono uniformemente sull'anello delle chiavi quindi con alta probabilità un nodo n qualsiasi è successore di una percentuale 1/N del totale delle chiavi, che sono 2^m. Dunque se ci sono x nodi maligni allora la frazione di chiavi catturate da un nodo tende a f = x/N, quindi la frazione di chiavi catturate complessive è fN = x. Nel caso di attacco Eclipse invece la percentuale di chiavi catturate cresce in modo più che lineare rispetto alla percentuale di nodi maligni. Questo per due ragioni:

a) I maligni catturano anche le chiavi non destinate a loro e non solo quelle a loro destinate. Questo è implementato nel seguente modo: quando un maligno riceve una chiave di cui non è successore, esso la reinstrada consultando la propria finger table contenente solo i nodi a cui collide. Perciò tale chiave terminerà il proprio path sul successor maligno di tale chiave e mai sul successor onesto di tale chiave.

b) I nodi maligni inquinando in modo attivo i nodi onesti, riescono a dirottare parte degli instradamenti dei nodi onesti verso i nodi maligni. L'inquinamento attivo riguarda due strutture dati Chord relativi all'instradamento: la finger table e la successor list.

Riassumendo vi sono due motivi per cui i nodi maligni catturano una percentuale di chiavi decisamente superiore rispetto al caso Sybil: i maligni inquinano i nodi onesti non solo in modo passivo ma anche in modo attivo: danno risposte maligne ad ogni fix_finger call che catturano e comunicano successor list completamente maligne ai nodi onesti. Come l'attacco Sybil, l'attacco Eclipse è un attacco distribuito in quanto è realizzato da più nodi maligni sparsi nella rete, tutti aventi lo stesso ruolo. Per quanto riguarda il comportamento inter nodi maligni possiamo affermare che l'attacco Sybil è non cooperativo in quanto i singoli nodi maligni sono ignari degli altri nodi maligni mentre l'attacco Eclipse è cooperativo perchè ogni nodo maligno collabora con gli altri maligni negli instradamenti (collusione). Sulle informazioni false prima citate, si sottolinea che mentre in Eclipse i nodi maligni forniscono informazioni false nella maggior parte dei casi, in Sybil i nodi maligni forniscono sempre risposte vere: questo perchè in Sybil i maligni continuano a seguire il protocollo originale laddove in Eclipse i maligni deviano dal comportamento originale al fine di rispondere con informazioni solo maligne. Tuttavia, seppure in maniera assai inferiore, anche in Sybil i nodi onesti sono soggetti ad inquinamento. Tale inquinamento è quello dovuto alle risposte vere maligne provenienti dai nodi maligni o dagli onesti inquinati. In Eclipse invece il grado di inquinamento è di gran lunga superiore a causa delle risposte false (quindi maligne) provenienti dai nodi maligni o dagli onesti inquinati.

4.2.3 Obbiettivo dell'attacco Eclipse

D’ora in poi, se non diversamente specificato, i nodi maligni eseguono l’attacco Eclipse e non semplicemente l’attacco Sybil. L'attacco Eclipse ha lo scopo di catturare il maggior numero possibile di messaggi da parte dei nodi maligni. Un messaggio catturato termina sempre su un maligno anziché su un onesto. Esistono due modalità di cattura: la prima, che è presente sia con Sybil che con Eclipse, si ha quando un messaggio termina sul suo vero successor e quest’ultimo è un nodo maligno. Questo tipo di cattura la chiamiamo passiva perché non è espressamente voluta dal software ma è il semplice effetto della presenza dei nodi maligni nella rete. La seconda, che è quella che dà luogo all’effetto eclissi e che dunque si ha solo con l’attacco Eclipse, si ha quando un messaggio il cui successor è un nodo onesto non arriva a tale nodo onesto, in quanto il messaggio viene “rubato” dal set dei nodi collusi.

Questo tipo di cattura la chiamiamo attiva perchè è il risultato del codice maligno. La cattura passiva e attiva dei messaggi da parte dei maligni può servire ad ottenere diversi effetti sui messaggi:

1) Eliminazione. I messaggi dell'applicazione distribuita sui nodi onesti non arrivano ai nodi onesti ma terminano sui maligni e di conseguenza si ha un degrado a livello applicativo.
2) Eavesdropping: i nodi maligni possono catturare i messaggi applicativi per leggerne il contenuto.
3) Tampering: i nodi maligni possono catturare i messaggi applicativi per modificarne il contenuto per ottenere un comportamento maligno a livello applicativo nei nodi onesti.

Ricordiamo il motivo del nome dell'attacco: l'eclissi sta nella capacità di un nodo maligno di partizionare la rete. Riferendoci all'anello Chord, un nodo maligno n' catturando tutti i messaggi che vi transitano, impedisce ai messaggi generati dagli onesti che precedono tale maligno n' di raggiungere i nodi che seguono n'. Di conseguenza il maligno n' provoca un effetto eclissi in quanto tende ad impedire la raggiungibilità dei nodi che lo seguono da parte dei nodi che lo precedono. Abbiamo detto "tende" ad impedire e non impedisce poiché in generale i nodi che precedono n' possono ancora raggiungere i nodi che lo seguono fintantoché il path dei messaggi non passa per n'.
Inoltre, a ben vedere, può esserci un altro scopo dell'attacco Eclipse. Esso può essere infatti la base per lanciare un DDoS (Distributed Denial of Service). Ad esempio, l'attaccante, attraverso del malware, può forzare i suoi nodi vittima ad entrare un in grosso overlay facendo comportare questi nodi vittima in modo che essi collidano fra essi. Questa botnet fornirebbe poi informazioni di routing colluse al resto dell'overlay in modo che quest'ultimo convogli il maggior traffico possibile ai nodi della botnet. Si osservi come, in questa visione dell'attacco Eclipse, quelli che usualmente chiamiamo nodi maligni (perché collusi) sono in realtà i nodi vittima.

4.2.4 Comportamento dei nodi maligni

Posizionamento nell'anello

I nodi maligni entrano ed escono dalla rete come nodi onesti ossia senza avere la possibilità di scegliere la loro chiave, la quale è sempre un intero uniformemente distribuito in $[0,2^{m-1}]$ risultante dall'hash dell'indirizzo IP del nodo. L'ipotesi sull'ingresso dei nodi nella rete è perciò identica al caso di attacco Sybil, ossia i nodi non hanno la possibilità di scegliere la loro chiave ovvero la loro posizione nell'anello delle chiavi. L'attaccante, se volesse posizionarsi ad un certo valore di chiave, dovrebbe trovare l'indirizzo IP il cui hash è pari a quella chiave. Nella plausibile ipotesi di adottare una comune funzione di hash crittograficamente sicura, la probabilità di successo di un attacco di preimmagine è trascurabile. E anche se l'attacco di preimmagine avesse successo, non è scontata la possibilità di assegnarsi o di farsi assegnare l'indirizzo IP trovato ad uno dei nodi che l'attaccante sta usando. Come vedremo più avanti, tuttavia, non è importante riuscire a trovare l'indirizzo IP che dia un dato valore di chiave, ma piuttosto è spesso sufficiente per gli scopi di posizionamento dell'attaccante, riuscire a trovare indirizzi IP che cadono all'interno di un intervallo di chiavi. Più è ampio tale intervallo e minore è il numero di tentativi che l'attaccante deve fare. Per evitare del tutto il rischio che l'attaccante si posizioni in una certa regione dell'anello occorre far sì che l'assegnamento della chiave del nuovo nodo avvenga in modo centralizzato tramite un supernodo fidato.

Azioni sui messaggi

Azioni dei nodi maligni si distinguono in due categorie:

1) Sui dati
2) Sul controllo

Con ciò si intendono le azioni malevoli sui messaggi di Chord che trasportano del payload applicativo, ovvero generato dall'applicazione distribuita che utilizza Chord come overlay routing (key based routing), o come supporto alla distributed hash table, ecc, in breve come servizio di overlay.

1) Per quanto concerne le azioni sui dati si distinguono tre casi:

1.1) Il messaggio applicativo ha una chiave di destinazione d il cui vero successore è il maligno n. In questo caso il messaggio applicativo termina sul maligno n e questo accade indipendentemente dal fatto che tale messaggio sia stato catturato o meno da nodi maligni. Questo comportamento è l'unico comportamento maligno che caratterizza l'attacco Sybil.
1.2) Il messaggio applicativo ha una chiave di destinazione d di cui n è il falso successore ma non il vero successore. Se il messaggio viene catturato da un nodo maligno n' allora quest'ultimo lo instrada secondo la propria finger table e successor list, le quali sono inquinate al 100%. Per questo motivo, un messaggio applicativo instradato su un maligno n' rimane intrappolato nel set dei nodi collusi e termina sul primo successore maligno della chiave del messaggio applicativo (il falso successore della chiave d). Se invece il messaggio non è catturato da nessun maligno, esso arriverà al suo vero successore. Il vero successore può essere un nodo maligno (caso 1.1) oppure un nodo onesto.
1.3) Il messaggio applicativo ha una chiave di destinazione d di cui n non è né il falso successore né il vero successore. In questo caso il nodo n si comporta esattamente come il nodo maligno n' descritto nel punto 1.2. La conseguenza è che il messaggio finirà sul primo maligno che succede la chiave d nell'anello.

Si osservi tuttavia che dal mero punto di vista dei nodi onesti, il comportamento dei maligni sopra esposto è identico al caso in cui i nodi maligni eliminano ogni messaggio applicativo che ricevono. Infatti in ambi i casi un messaggio catturato dai maligni non arriva mai al successore onesto della chiave del messaggio e di conseguenza i maligni hanno lo stesso effetto di DoS.

2) Vediamo ora le azioni maligne sul controllo:

Con azioni sul controllo si intendono azioni malevoli che coinvolgono messaggi Chord che trasportano informazioni del protocollo Chord stesso, ossia informazioni di controllo di livello overlay. Queste sono le azioni che danno origine all'inquinamento dei nodi onesti cioè la presenza di informazioni Chord maligne nei nodi onesti.

2.1) Inizializzazione del successor, predecessor e successor list

Analogamente al caso delle stabilize & notify, l’idea di base da parte dei nodi maligni è quella di memorizzare e perciò fornire i veri predecessor e successor ma di fornire una successor list maligna. Comunicare i veri predecessor e successor, come detto prima, è utile a garantire che gli onesti non abbiano un falso predecessor. Se ciò accadesse, gli onesti catturerebbero chiavi ovvero farebbero terminare messaggi di cui non sono responsabili.

Nelle nostro ipotesi invece, è l'attaccante stesso che aspira a rubare il maggior numero di messaggi possibile. Naturalmente, in un ottica di DoS, l'attaccante potrebbe essere soddisfatto anche creando situazioni in cui gli onesti catturano messaggi ma quest'ultimo scenario, oltre a non massimizzare i messaggi catturati dall'attaccante, potrebbe anche sfuggire facilmente al suo controllo. Per questo motivo siamo nell'ipotesi in cui l'attaccante è interessato a far sì che gli onesti si comportino in modo corretto.

Quando un nodo n entra nella rete contatta immediatamente un nodo già presente nella rete (bootstrap node) tramite un messaggio di join call. Esso risponde con una join response dando la possibilità al nuovo nodo n di inizializzare il puntatore al suo successor, predecessor e la successor list. Il meccanismo di join call and response è più comunemente chiamata join.

Distinguiamo tre casi di join:

2.1.1) Quando un nodo onesto n entra nell'anello, se contatta con la join call un bootstrap node onesto n', quest'ultimo nodo fa 2 operazioni: 1) Risponde ad n (join response) comunicandogli la successor list di n': così il nodo n quando riceve la join response, inizializza la sua successor list prendendo la successor list ricevuta da n', dopo aver scartato l'ultima entry, e inserito n' come prima entry (fifo). Notare che n, avendo inserito n' come prima entry della successor list, ha assunto n' come suo successor. 2) Se la successor list di n' è vuota, vi inserisce chi ha fatto la join call (il nodo n).
Se si usa anche la aggressive join (proposta di Oversim), alla procedura precedente se ne aggiungono altre due simultaneamente: 1) Il bootstrap node n' che riceve la join call, prende n come suo predecessor e manda un messaggio newsuccessorhint al suo vecchio predecessor contenente il nuovo predecessor di n' (che è n). Il vecchio predecessor di n' aggiorna dunque ad n il suo nuovo successor se n cade fra il vecchio predecessor e il successor del vecchio predecessor. 2) Il nodo n che riceve la join response aggiorna il suo predecessor al vecchio predecessor del bootstrap node n' (perciò la join response contiene anche il vecchio predecessor di n' e non solo la successor list di n').
2.1.2) Se invece un nodo (onesto o maligno) n entra nell'anello e contatta come bootstrapnode n' un nodo maligno, quest'ultimo nodo fa le due operazioni già descritte: 1) Risponde ad n (join response) comunicandogli la successor list di n' (tutta maligna): così il nodo n quando riceve la join response, inizializza la sua successor list prendendo la successor list ricevuta da n', dopo aver scartato l'ultima entry, e inserito n' come prima entry (uguale al caso 2.1.1). Come al caso 2.1.1, il nodo n avendo inserito n' come prima entry

della successor list, ha assunto n' come suo successor. 2) Se il vero successor di n' non è ancora stato inizializzato, esso viene aggiornato a chi ha fatto la join call, cioè il nodo n (uguale al caso 1).
Esattamente come i casi 1 e 2, se si usa anche la aggressive join (proposta di Oversim), alla procedura precedente se ne aggiungono altre due simultaneamente: 1) Il nodo n' che riceve la join call, prende n come suo predecessor e manda un messaggio newsuccessorhint al suo vecchio predecessor contenente il nuovo predecessor di n' (che è n). Il vecchio predecessor di n' aggiorna dunque ad n il suo nuovo successor se n cade fra il vecchio predecessor e il successor del vecchio predecessor. 2) Il nodo n che riceve la join response aggiorna il suo predecessor al vecchio predecessor del bootstrap node n' (quindi se è abilitata la aggressive join allora la join response contiene anche il vecchio predecessor di n' e non solo la successor list di n'). In definitiva si osserva che qui l'aggressive join è identica al caso 2.1.1.
Come si nota, un nodo maligno n' che riceve una join call da un nodo n (n' è il bootstrap node di n) esegue lo stesso identico protocollo che un nodo onesto eseguirebbe. In definitiva, il bootstrap node maligno n' agisce in modo da memorizzare le informazioni vere per quanto riguarda i suoi veri successor e predecessor così da fornirle agli altri nodi nelle operazioni di join, stabilize e notify. L'unica informazione possibilmente falsa che n' elargisce al nodo n è la sua successor list (eccetto la prima entry che corrisponde al successor), infatti questa verrà usata da n per i suoi lookup.

2.1.3) Un nodo maligno n che entra nell'anello costruisce immediatamente la sua successor list e la sua finger table senza usare la notify e la fix finger rispettivamente, infatti nella nostra ipotesi ogni maligno conosce ogni altro maligno. Come un nodo onesto, quando un nodo maligno n entra nell'anello, contatta un bootstrap node n' (onesto o maligno) dal quale riceve la join response e dopodiché il nodo n: 1) Ignora la successor list di n' ricevuta con la join response perché se la costruisce da sé con la lista locale dei nodi maligni (diverso dagli onesti) 2) Inizializza il suo vero successor ad n' (come gli onesti).
Se si usa l'aggressive join il nodo n aggiorna il suo vero predecessor al valore di predecessor che il bootstrap node n' aveva prima di ricevere la join call (come gli onesti).

2.2) Mantenimento del successor, della successor list e del predecessor

In Chord sono definite due funzioni chiamate stabilize e notify che si occupano di aggiornare periodicamente il puntatore al successor, al predecessor e la successor list. L'aggiornamento è necessario perchè in generale i nodi possono entrare ed uscire dalla rete individualmente e non tutti simultaneamente (churn). In particolare applicazioni dove invece i nodi sono statici

(no churn) le funzioni stabilize e notify sono perfettamente inutili ed è ragionevole disattivarle per ridurre il traffico di controllo.

Un maligno n mantiene il suo vero successor s (maligno o onesto che sia) cosicché, il maligno n, per aggiornare il suo vero successor, manda la stabilize call al suo vero successor s: quest'ultimo risponde ad n con il predecessor p di s, il quale diventa il nuovo successor di n se p è compreso fra n ed s. Il fatto che n mandi la stabilize call al suo vero successor s massimizza la probabilità che il p comunicato da s a n cada tra n ed s e dunque è un migliore successor di s per n. In altre parole, anche il nodo maligno n vuole sapere chi è il suo vero successor s. Questo è necessario affinché il nodo s abbia il predecessor corretto, infatti la conoscenza del vero successor s di n consente ad n di mandare la notify call al suo vero successor s cosicché s potrà aggiornare il suo predecessor al suo vero valore e cioè n se n cade fra il corrente predecessor di s ed s. E' infatti essenziale che un nodo s conosca il suo vero predecessor altrimenti se s è onesto e il suo predecessor non fosse corretto (cioè più lontano di quello vero) si troverebbe a catturare messaggi.

Il maligno n mantiene anche il vero predecessor p (maligno o onesto che sia) in modo da rispondere correttamente alla stabilize call da parte di un nodo n' che precede n. In questo modo si assicura che n' aggiorni il suo successore al vero valore p. In questo modo il nodo n' in seguito contatta con la notify call il suo vero successor (il nodo p ricevuto dal maligno n) cosicché p aggiorna il suo predecessor al valore corretto ovvero al nodo n' se n' cade fra p e il suo corrente predecessor. Come già detto, la correttezza del predecessor di un onesto p è necessaria affinché l'onesto non catturi chiavi.

Il maligno n risponde alla notify call di un nodo n' con la sua successor list maligna. Chord specifica che non appena n' riceve la successor list di n (notify response), il nodo n' elimina la entry più lontana di tale successor list ed esattamente come una coda fifo, vi aggiunge all'inizio il nodo n da cui ha ricevuto la successor list (in altre parole un nodo che riceve una notify response, assume che essa arrivi dal suo successor infatti la prima entry della successor list è per definizione il successor). Il nodo n' quindi, avendo contattato il maligno n tramite la notify call, si ritrova ad avere il corretto successor (corretto perchè non cambia con la notify response che invece è maligna) mentre il resto della successor list è tutta maligna, dunque verosimilmente falsa. Infatti, la successor list è usata solo per i lookup e dunque è nell'interesse dell'attaccante che sia maligna.

2.3) Inizializzazione e mantenimento della finger table e della successor list del nodo maligno

Un nodo maligno n utilizza la propria finger table e successor list esattamente come un nodo onesto ossia per instradare i messaggi applicativi e le fix finger calls. La sola differenza con i nodi onesti sta nell'inizializzazione e nel mantenimento di tali informazioni. Per quanto concerne la finger table un nodo onesto la inizializza e la aggiorna tramite la funzione fix finger mentre un nodo maligno la inizializza e la aggiorna consultando la lista dei nodi maligni. Per quanto riguarda la successor list, laddove un nodo onesto la inizializza tramite la join response e la aggiorna con la notify response, un nodo maligno la inizializza e la aggiorna consultanto la lista dei nodi maligni. La conseguenza di tutto ciò è che la finger table e la successor list dei maligni è come se fossero quelle degli onesti nel caso in cui nella rete vi siano solo i nodi maligni. Per questo motivo la finger table e la successor list di un maligno ha due proprietà: a) Rispetta appieno il procollo Chord b) Contiene esclusivamente riferimenti a nodi maligni (secondo l'ottica dei nodi onesti è come dire che è inquinata al 100%).

2.4) Inquinamento della finger table dei nodi onesti

La diffusione di riferimenti a nodi maligni nelle finger table è senza dubbio la peculiarità dell'attacco Eclipse. I nodi maligni inquinano in modo passivo e attivo le finger table dei nodi onesti. L'inquinamento passivo, il quale si ha anche nel caso Sybil, è quello dovuto dalla sola presenza nella rete dei nodi maligni. E' un inquinamento naturale e non è indotto da nessuna sorta di intelligenza da parte dei nodi maligni, infatti nel Sybil i maligni seguono lo stesso identico protocollo Chord degli onesti. Per il semplice fatto che i nodi maligni sono nodi, essi compariranno necessariamente nelle finger table in certi nodi onesti secondo le normali regole di Chord. Se nella rete vi è una frazione f di nodi maligni allora il solo inquinamento passivo nei nodi onesti è ancora f. Dal punto di vista operativo l'inquinamento passivo delle finger table avviene nel seguente modo: quando un nodo onesto n esegue la fix finger call per trovare una finger i ($0 \leq i \leq m - 1$), se capita che le chiavi dei nodi nella rete sono tali che il successor del valore start s della fix finger call è un nodo n' maligno ecco che il nodo onesto n avrà la finger i maligna, cioè inquinata. Questo non è inquinamento attivo perchè è vera l'affermazione "il nodo maligno n' è il successor del valore start s". Se all'opposto fosse falsa allora si parlerebbe di inquinamento attivo. L'inquinamento passivo esiste in Sybil e dunque anche in Eclipse. Se aggiungiamo anche l'inquinamento attivo (presente in Eclipse ma non in Sybil) allora l'inquinamento totale è maggiore di f dato che a quello passivo si somma quello attivo. Quest'ultimo è il frutto del comportamento malevolo dei nodi maligni

ossia della loro deviazione dal protocollo con l'esplicito obbiettivo di fornire il maggior numero possibile di informazioni maligne ai nodi onesti. Operativamente l'inquinamento attivo della finger table avviene nel seguente modo: un nodo maligno n' avendo una finger table e una successor list 100% maligne (perchè sono attivamente autoinquinate), farà si che ogni fix finger call che riceve e di cui non è successor del valore start s, venga assorbita senza uscita dal sottoinsieme dei soli nodi maligni. A questo punto la fix finger call terminerà sul successor maligno del valore start s, il quale, se non coincide con il vero successor, dà luogo ad una fix finger response falsa. L'inquinamento attivo delle finger table degli onesti permette di deviare gran parte del traffico in uscita dagli onesti verso i (pochi) nodi maligni. L'attaccante, in prima battuta, non ha il controllo sulle deviazioni dei flussi di messaggi poiché i nodi maligni come avevamo detto non possono scegliere la loro posizione nell'anello delle chiavi. Se ad esempio l'attaccante fosse interessato a bloccare tutti i messaggi verso un nodo onesto n (oscuramento completo in ingresso), esso dovrebbe piazzare i propri nodi maligni in modo da inquinare opportunamente tutti quei nodi onesti che, secondo il protocollo, dovrebbero avere il nodo n nella finger table e nella successor list. Di posizioni strategiche di questi nodi maligni ce ne sono in generale tante ma quella che richiede meno sforzo vede l'inserimento di un nodo maligno n' per ogni nodo onesto p che dovrebbe avere il nodo onesto n nella finger table e/o successor list. Ogni nodo maligno n' dovrebbe entrare nella rete come successor di ogni nodo p così da essere sicuri che ogni nodo p sia inquinato al 100% e non abbia quindi nessun puntatore al nodo n. L'attaccante allora, per ogni nodo p, a partire da un set di indirizzi IP random, dovrebbe trovarne uno il cui hash dà una chiave compresa tra il nodo p considerato e il relativo corrente successor onesto. Quest'ultimo lo possiamo denominare come problema di posizionamento del successor. Nell'ipotesi che l'hash sia uniforme in $[0,2^{m-1}]$ allora la probabilità di successo di un tentativo di soluzione di tale problema è pari a 1/N, infatti distanza_media_nodi/circonferenza_anello = $((2^m)/N)/2^m = 1/N$. Se invece l'intento dell'attaccante è quello di catturare tutto il traffico in uscita da un nodo onesto n, allora deve far sì che la finger table e la successor list del nodo n siano inquinate al 100%. Questo richiede uno sforzo inferiore rispetto al caso precedente poiché è sufficiente che solo il successor di n sia maligno. L'attaccante ha quindi un solo problema di posizionamento del successor da risolvere anziché tanti quanti erano i nodi p nel caso di oscuramento completo in ingresso. Va riconosciuto che entrambi i casi appena descritti di cattura totale mirata ad un nodo (oscuramento completo in ingresso e in uscita) sono dei casi particolari di attacchi Eclipse poiché il posizionamento dei nodi maligni non è random uniforme sull'anello come avviene invece nell'attacco Eclipse classico, bensì è mirato in certi settori. A parte il problema di tentare indirizzi IP random per trovare la chiave desiderata, l'oscuramento in uscita non ha difficoltà teoriche da parte dell'attaccante: è sufficiente trovare

un indirizzo IP il cui hash è una chiave che cade fra il nodo vittima n e il suo corrente successor onesto. Assolutamente più arduo è realizzare l'oscuramento in ingresso di un nodo n dal momento che i nodi maligni non conoscono a priori quali sono i nodi onesti p che dovrebbero avere il nodo n presente nella loro finger table e/o successor list. L'attaccante allora, al fine di scoprire tali nodi onesti p, può inizialmente entrare nella rete con nodi distribuiti random e memorizzare tutte le chiavi sorgenti (che corrispondono per forza a nodi) dei messaggi nella speranza che queste chiavi esauriscano l'insieme dei nodi p. Per ognuna di queste chiavi (che corrispondono a nodi onesti) i nodi maligni possono ricostruire le relative finger table corrette (tramite fix finger) e vedere se il nodo n è presente come finger. Una tale chiave (nodo) è un nodo p. In definitiva l'attacco a Chord ha un effetto fortemente quantitativo ma un po' meno qualitativo nel senso che è in grado di arrecare un grosso danno con poco sforzo ma allo stesso tempo ha una limitata capacità di indirizzare il danno a specifici nodi onesti.

Vediamo meglio come evolve l'inquinamento. L'inquinamento dei nodi onesti evolve in due fasi: 1) Nella fase di bootstrap, cioè quando un nodo entra nella rete ed esegue la funzione di join e di fix_finger 2) Durante il tempo di vita del nodo nella rete nei momenti in cui vengono eseguite le funzioni di mantenimento periodiche dei puntatori al successor, predecessor, successor list (stabilize e notify) e per il mantenimento della finger table (fix_finger).

Nell'ipotesi che tutti i nodi maligni entrano nella rete nello stesso momento (cioè nel giro di pochi secondi) e che non vi sia churn né per i nodi onesti né per i nodi maligni, allora il grado di inquinamento medio sui nodi maligni si stabilizza nel tempo ad un valore di regime.

Infine, a parita di percentuale di nodi maligni, per reti più grandi la percentuale di chiavi catturate cresce. Ciò si spiega nel fatto che all'aumentare di N cresce pure la lunghezza media dei path e con essa anche la probabilità che il path finisca su un nodo maligno.

Nel grafico seguente valutiamo l'effetto ultimo dell'attacco Eclipse, ossia quello di catturare messaggi. Il grafico si riferisce ad una rete da N = 100 nodi. Si osservi come, anche con piccole percentuali di nodi maligni, il set di nodi collusi è in grado di venire in possesso della maggior parte del traffico nell'overlay. Questa caratteristica, come già accennato, è anche un'ottima opportunità per un DDoS.

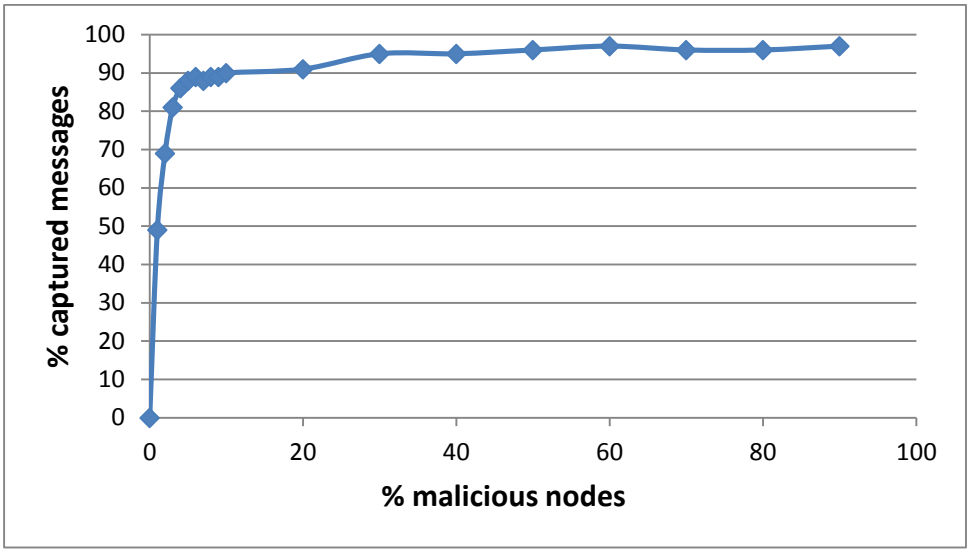

Fig. 14: *Andamento della percentuale di messaggi catturati in funzione della percentuale di nodi maligni per un overlay da N = 100 nodi.*

4.3 Ambiente sperimentale

4.3.1 Simulazioni di rete e difese

Le simulazioni di rete (Chord overlay) sia senza attacco sia con attacco Eclipse e con le difese, sono state eseguite per mezzo del simulatore open source OverSim. Abbiamo modificato il codice sorgente (C++) di OverSim per introdurre tre nuove caratteristiche:

1) Raccolta dei dati statistici di interesse sia per l'identificazione dell'attacco (attributi) che per le difese (% chiavi catturate) e più in generale dati legati al funzionamento dell'overlay Chord (ad es. stampa finger table, successor list, ecc.).
2) Implementazione attacco Eclipse in Chord. Abbiamo implementato l'attacco come è stato descritto in questo capitolo, quindi semplificando i nodi collusi differiscono dagli onesti dal fatto che vedono l'overlay popolato solo dai nodi a cui colludono.
3) Implementazione delle difese. Tutte le difese che saranno descritte nel capitolo sulle contromisure le abbiamo implementate integrandole direttamente nel codice del simulatore OverSim. Le modifiche hanno interessato le classi comuni ai diversi protocolli overlay (OverSim/src/common) e le classi specifiche di Chord (OverSim/src/overlay/chord). Non abbiamo né creato nuove classi né nuove librerie ma semplicemente modificato le classi già esistenti. Questo perchè la maggior parte delle difese non sono altro che delle versioni

diverse delle funzioni già previste. Non sono state effettuate modifiche ad Omnet++ poiché l'interesse di questa tesi è focalizzato sull'overlay layer, il quale non è presente in Omnet++, ma solo in OverSim.

Oversim è un simulatore open source di reti overlay che si appoggia al simulatore di reti OMNet++. Sviluppato dall'Università Tecnica di Budapest, OMNet++ è un ambiente di simulazione ad eventi discreti. Sia OMNet++ che Oversim sono scritti in C++. Oversim invece è stato sviluppato dall'Istituto di Tecnologia di Karlsruhe (Germania) come parte del progetto ScaleNet finanziato dal ministero federale tedesco dell'istruzione e della ricerca. Oversim comprende l'implementazione dei più comuni protocolli overlay strutturati (Chord, Kedemlia, Pastry) e non strutturati (GIA).

Di seguito sono elencate le caratteristiche salienti di Oversim:

- *Flexibility:* Oversim consente la simulazione di overlay strutturati e non strutturati (al momento Chord, Pastry, Bamboo, Koorde, Broose, Kademlia, GIA, NICE, NTree, Quon, Vast, e Publish-Subscribe for MMOGs sono implementati). La modularità del disegno di Oversim e l'uso ricorrente delle API Comuni rendono Oversim un ambiente orientato allo sviluppo, quindi per aggiungere e/o modificare i protocolli.
- *Interactive GUI:* Oversim si avvale della GUI di OMNet++ per visualizzare la topologia overlay, i messaggi e le variabili di stato dei nodi. Può essere utile anche per validare le modifiche o i nuovi protocolli.

- *Exchangeable Underlying Network Models:* per quanto riguarda la rete underlay, vi è la possibilità di utilizzare sia un modello più realistico (INETUnderlay) che un modello semplificato, (SimpleUnderlay), quindi meno impegnativo per le risorse, per consentire simulazioni di larga scala.

- *Scalability:* Oversim è stato pensato per essere performante. Ad esempio su un moderno desktop pc una rete Chord da 10000 nodi può essere simulata in real-time.

- *Base Overlay Class:* la classe BaseOverlay si presta allo sviluppo di reti peer-to-peer strutturate, grazie alle RPC (Remote Procedure Calls), alla classe di loookup generica e le API comuni per gestire la comunicazione anche verso le applicazioni.

- *Different Routing Modes:* la suddetta classe BaseOverlay offre diverse modilatità di

routing: iterative, exhaustive-iterative, semi-recursive, full-recursive e source routing recursive.

- *Reuse of Simulation Code:* le implementazioni dei protocolli overlay sono riutilizzabili per applicazioni reali, in modo ad esempio di poter utilizzarle sulla rete reale di test PlanetLab. Pertanto, il codice di Oversim è in grado di lavorare con pacchetti dati reali.

- *Churn Models:* sono previsti diversi modelli di churn (tempi di vita di nodo con distribuzione di Weibull, Pareto, etc) e in ogni caso è facile aggiungere modelli di churn ad hoc.

- *Statistics:* vi sono diverse funzioni per la raccolta dei dati statistici come ad esempio numero di messaggi inviati, ricevuti, transitati, hop count, ecc. Sono inoltre inclusi alcuni Python scripts per fare post-processing sui dati raccolti e per generare grafici.

- *Applications:* Oversim non implementa soltanto l'overlay layer ma pure le più tipiche applicazioni distribuite com Distributed Hash Table (DHT), Internet Indirection Infrastructure (i3), Scribe, SimMud, P2PNS e applicazioni di test (KBRTestApp, DHTTestApp, e SimpleGameClient).

4.3.2 Identificazione Attacco

Per quanto riguarda l'identificazione della presenza / assenza di attacco, ci siamo avvalsi del software Weka (University of Waikato, New Zealand). Scritto in Java e open souce, Weka è una collezione dei più comuni algoritmi di machine learning. Esso consente data pre-processing, classificazione, regressione, clustering, regole di associazione e visualizzazione di dati. Nel nostro caso Weka ci è stato utile per la sua funzionalità di classificazione ma non abbiamo dovuto modificare il codice sorgente.

Per ogni istante di tempo t in cui vogliamo valutare l'accuratezza di identificazione di presenza/assenza di attacco, diamo a Weka un file testuale di tipo .csv (test set) in cui ogni linea è il valore in t degli attributi di un certo nodo. Di conseguenza la percentuale di identificazioni corrette (accuratezza) all'istante t è la percentuale di nodi che identificano correttamente all'istante t l'assenza / presenza di attacco.

4.4 Valori comuni in tutte le simulazioni

I parametri comuni in tutte le simulazioni i cui risultati sono riportati in questa tesi, sia per l'identificazione che per le difese, sono i seguenti:

- Tempo di simulazione = 5500 s
- Tempo di raccolta dati: 5000s (intervallo da 500 s a 5500 s)
- Frequenza generazione messaggi per nodo = 0.2 messaggi/s (dunque un nodo nel tempo di raccolta dati genera 0.2·5000 = 1000 messaggi)
- Dimensione successor list (succlist_size) = 16 nodi
- Intervallo di fix finger = 100 s
- Intervallo di stabilize = 20 s

5
IDENTIFICAZIONE DELL'ATTACCO ECLIPSE

5.1 Introduzione

In questo capitolo proponiamo una soluzione al problema di identificare l'attacco Eclipse in un rete overlay Chord. Ricordiamo come abbiamo implementato l'attacco Eclipse al protocollo Chord: i nodi collusi rispondono alle richieste di informazioni di routing da parte dei nodi non collusi esattamente da protocollo, con l'unica differenza che i nodi collusi tengono informazioni di routing riguardanti soltanto gli altri nodi collusi. Di conseguenza, i nodi collusi rispondono ai nodi onesti soltanto con informazioni riguardanti i nodi collusi e inoltrano i messaggi applicativi provenienti dai nodi onesti solo agli altri nodi ai cui collidono. Facendo leva su questo comportamento anomalo dei nodi collusi, vedremo come i nodi onesti possano riconoscere l'assenza o la presenza dell'attacco analizzando le proprie informazioni di routing e/o i messaggi ricevuti.

L'identificazione dell'attacco ha due obbiettivi:

1) Avvisare l'applicazione che è in corso un attacco alla rete overlay che l'applicazione stessa vorrebbe utilizzare.
2) Attivare le difese: siccome gli algoritmi di difesa in generale determinano un overhead computazionale, di memoria e di traffico, in linea di principio è ragionevole attivarle solo se ha luogo l'attacco.

5.2 Proposta

Il problema di identificazione dell'attacco è un problema di classificazione binaria (attacco e non attacco). Per risolvere questo problema di classificazione binaria abbiamo optato per una soluzione di supervised machine learning, più precisamente ci siamo avvalsi degli alberi decisionali di classificazione. Si parla di machine learning di tipo supervised quando il modello viene costruito a partire da dati già classificati (training set). La soluzione qui proposta è inoltre di tipo distribuito, infatti ogni nodo esegue l'algoritmo di identificazione senza la cooperazione di nodi estranei a Chord. E' inoltre una proposta di tipo locale perché non richiede esplicitamente informazioni da altri nodi dell'overlay Chord. Queste due caratteristiche la rendono scalabile con la dimensione N dell'overlay.

La bontà di un sistema di classificazione di supervised learning dipende da:

1) Algoritmo scelto per costruire il modello (per esempio l'algoritmo C4.5)
2) Scelta degli attributi che costituiscono il vettore input dell'algoritmo
3) Qualità del training set con cui è stato costruito il modello

5.2.1 Modello

Come accennato il modello di classificazione che abbiamo utilizzato è un albero decisionale di classificazione, dunque un modello di classificazione di tipo supervised, ossia generato dall'applicazione di un algoritmo su istanze classificate a priori (training set). Un training set infatti nel nostro caso consiste in un insieme di istanze, le quali sono matematicamente dei vettori di F colonne. Un tale vettore è composto da F – 1 attributi (features) e da 1 classe (class, category, label). L'algoritmo di costruzione del modello costruisce il modello, nel nostro caso un albero, esaminando la correlazione fra gli F – 1 attributi e la classe delle istanze costituenti il training set. Il risultato è il modello di classificazione, ovvero un algoritmo che ha imparato a stimare la classe di un'istanza analizzando i suoi F – 1 attributi o una parte di essi. In questo caso specifico, un'istanza è un campione dello stato della rete vista in un dato momento da un generico nodo della rete. Per stato della rete di un nodo n all'istante t intendiamo l'insieme degli F – 1 attributi che il nodo n misura all'istante t. Gli attributi non sono altro che valori numerici che il nodo estrae delle proprie informazioni di routing e/o dall'osservazione dei messaggi ricevuti.
Come algoritmo di costruzione (training) dell'albero ci siamo avvalsi del noto algoritmo C4.5

[14]. Quest'ultimo è una versione migliorata dell'algoritmo ID3.

Algoritmo ID3

L'Iterative Dichotomizer 3 (ID3) è un algoritmo che costruisce un albero decisionale a partire da un training set nel seguente modo: ad ogni iterazione l'algoritmo divide le istanze del training set in due partizioni in funzione dell'attributo che, nell'iterazione corrente, è caratterizzato dal massimo information gain. Ogni nodo (iterazione) dell'albero rappresenta quindi la valutazione dell'attributo che nella partizione del subset ottenuta nel suo nodo padre ha il massimo information gain. L'information gain di un attributo in un determinato subset è un indice numerico che descrive la capacità di un attributo di discriminare le classi nelle istanze subset corrente. In altre parole, ad ogni partizionamento di un subset, viene scelto l'attributo che meglio caratterizza le istanze di quel subset in funzione della loro classe. La ricorsione termina quando un nodo è una foglia ovvero quando il subset è costituito interamente da istanze appartenenti ad una classe.

Algoritmo C4.5

Funziona esattamente come l'algoritmo ID3 ma con l'aggiunta dei seguenti miglioramenti:

1) Può trattare anche attributi continui anziché solo discreti
2) Contempla anche il caso in cui alcuni attributi delle istanze sono mancanti
3) E' possibile assegnare pesi diversi agli attributi
4) Consente il pruning dell'albero

Nel nostro caso sono utili i miglioramenti 1,2,4. Il primo è necessario poiché, come vedremo, tutti gli attributi scelti sono numeri reali quindi attributi continui. Il secondo caso è utile quando alcuni attrbuti di alcune istanze non sono definiti e questo può capitare in quelle realizzazioni, cioè in quelle particolari istanze, dove alcuni attributi non sono calcolabili (ad es. logaritmo di zero, divisione per zero, ecc.). La quarta caratteristica è importante per non avere un albero troppo profondo e/o sbilanciato in modo così da non avere un albero troppo gravoso in termini di memoria e computazionali per quanto riguarda il suo uso ovvero della classifcazione delle istanze. L'altro vantaggio del pruning è quello di contrastare l'overfitting, ovvero la costruzione di un albero con un tasso di errore molto basso per le istanze dalle quali è nato (training set) ma più elevato per le istanze non presenti nel training set.
Il modello scelto per monitorare la presenza / assenza di attacco è un albero decisionale

sviluppato dall'algoritmo C4.5. Vediamo adesso quali attributi abbiamo scelto per caratterizzare i training set dai quali nascono tali alberi decisionali. Per quanto riguarda gli attributi, affrontiamo ora due questioni: come vengono raccolti nel tempo e quali di essi vengono effettivamente scelti dall'algoritmo di costruzione dell'albero (algoritmo C4.5).

5.2.2 Attributi

In generale un modello di classificazione prende in input il vettore dell'istanza che si vuole classificare e si ottiene come risultato la classe a cui l'istanza appartiene secondo tale modello. Questa idea applicata all'identificazione dell'attacco in una rete si traduce in quanto segue: un nodo colleziona in tempo reale i dati che servono a calcolare gli elementi del suddetto vettore (attributi o features). Periodicamente, quando deve avvenire il test di identificazione dell'attacco, viene eseguito l'algoritmo di classificazione binaria che prende in input il vettore restituendo la sua classe (attacco o non attacco). Nella nostra proposta, l'aggiornamento del vettore avviene nel seguente modo: dividiamo il tempo in round r = 1,2,3... di durata T_r in cui ad ogni fine round viene aggiornato il vettore. Dopodiché ogni attributo del vettore viene aggiornato inizializzandolo al valore del round presente mediato con i valori che tale attributo aveva assunto negli ultimi W - 1 round. In questo modo si tiene conto della storia passata fino a WT_r secondi nel passato e questo aiuta a smussare eventuali picchi casuali. In altre parole, ogni attributo ha una sua sliding window di ampiezza W. Certamente questo riduce la reattività del sistema di identificazione al cambiare della situazione attacco / no attacco. Non abbiamo tuttavia compiuto un'indagine su questi scenari dinamici. E' vero che gli attributi sono aggiornati ad ogni fine round, tuttavia essi sono validi in ogni momento, quindi è possibile eseguire in qualsiasi momento il test di identificazione. Se scegliamo, come abbiamo fatto nella nostra implementazione, di aggiornare gli attributi ad ogni fine round, si ottiene un aggiornamento periodico di periodo T_r.

L'albero decisionale ideale rispetto ai diversi scenari di rete è tale da classificare con accuratezza superiore ad una certa soglia prefissata i diversi scenari. Tanto più uno scenario di rete si scosta dagli scenari contenuti nel training set e maggiore è il rischio che la classificazione mostri una bassa accuratezza. Gli attributi devono dipendere il più possibile dalla percentuale di nodi maligni e il meno possibile da qualsiasi altro fattore che non sia la percentuale di nodi maligni (fattori di disturbo). Nel nostro caso i diversi scenari dipendono dalle diverse condizioni di churn, da quanto traffico è generato dalle applicazioni e a quali nodi è indirizzato e dalla dimensione dell'overlay (numero di nodi N). Tutte queste variabili

indipendenti e che influenzano la classificazione sono fattori di disturbo della classificazione. Non abbiamo valutato la correlazione fra gli attributi e tutti fattori di disturbo che possono influenzarli perchè un lavoro esaustivo richiederebbe molto tempo. Abbiamo però valutato l'andamento fra gli attributi e il fattore di disturbo dato dal numero di nodi N. Naturalmente un attributo è tanto migliore rispetto ad N tanto più non dipende da N.

Nelle pagine seguenti elenchiamo ed analizziamo i 14 attributi che abbiamo preso in esame. Per ognuno di essi diamo tre informazioni:

1) Calcolo nel tempo (round r): è la formula con cui il software di monitoring dell'attacco calcola ad ogni round r l'attributo.
2) Valore medio (senza attacco): è il valore medio dell'attributo in condizioni che rispettano il protocollo, quindi nel caso senza attacco. E' lo scostamento dell'attributo dal suo valore di aspettazione che è utile alla segnalazione dell'attacco.
3) Grafico: mettiamo congiuntamente in evidenza la dipendenza di ogni attributo rispetto al numero di nodi N (Overlay Size) e rispetto alla classe (attacco o assenza di attacco). Nel caso di attacco abbiamo sempre tenuto una frazione di nodi maligni pari a f = 0.02.

Prima di elencare gli attributi è utile leggere la seguente legenda:

d = chiave del nodo sorgente della risposta (si veda il par. "Termini e notazioni" per il significato di risposta)
current_node = chiave del nodo che effettua la misura dell'attributo
hop_count(k,i) = hop count del messaggio ricevuto i durante il round k
k = indice di round
K = chiave di destinazione del messaggio
W = lunghezza in numero di round della sliding window
last_finger(k) = finger più lontana dal nodo corrente durante il round k
$N_R(k)$ = numero di risposte ricevute durante il round k
$N_f(k)$ = numero di entry della finger table durante il round k (è a sua volta un attributo, che ridenominiamo ft_length)
$N_{mf}(k)$ = numero di messaggi inoltrati ad una finger durante il round k
$N_s(k)$ = numero di successor list entry (succlist_size)
N_{mr} = numero di messaggi ricevuti durante il round k
Elenco attributi:

1) response_distance

Calcolo nel tempo (round r):

$$response\_distance(r) = \frac{1}{W} \sum_{k=r-W+1}^{r} \left(\frac{1}{N_R(k)} \sum_{i=1}^{N_R(k)} (d(k,i) - K(k,i)) \right) \quad mod\ 2^m$$

Valore medio (senza attacco):

$$E[response\_distance] = \frac{2^m}{N}$$

In [15] viene fatto notare che, se N è elevato, la distanza fra un punto casuale su una circonferenza e il punto casuale successivo è approssimabile ad una variabile casuale geometrica di media pari alla distanza media fra tali punti casuali. Siccome le chiavi di destinazione dei nodi sono punti casuali sull'anello, allora una response_distance vera, ossia la distanza fra la chiave k di destinazione di un messaggio e il relativo successor d è una variabile casuale identica alla distanza fra nodi. Infatti anche i nodi e dunque il successor d sono punti casuali sull'anello. Sappiamo inoltre che la distanza media fra nodi è $2^m/N$. Al contrario se response_distance è falsa, ossia la risposta non proviene dal vero successor del messaggio, allora essa è una realizzazione della distanza fra nodi maligni, che è una geometrica di media $2^m/fN$ che quindi è 1/f volte maggiore del caso di distanza media fra nodi. Qui supponiamo che la distanza fra maligni sia ancora geometrica perché ipotizziamo, come è verosimile, che anche i maligni sono uniformi sull'anello. Ricapitolando, se response_distance è vera allora essa è la realizzazione di una geometrica di media $2^m/N$, se è falsa è invece una realizzazione di una geometrica di media $2^m/fN$.

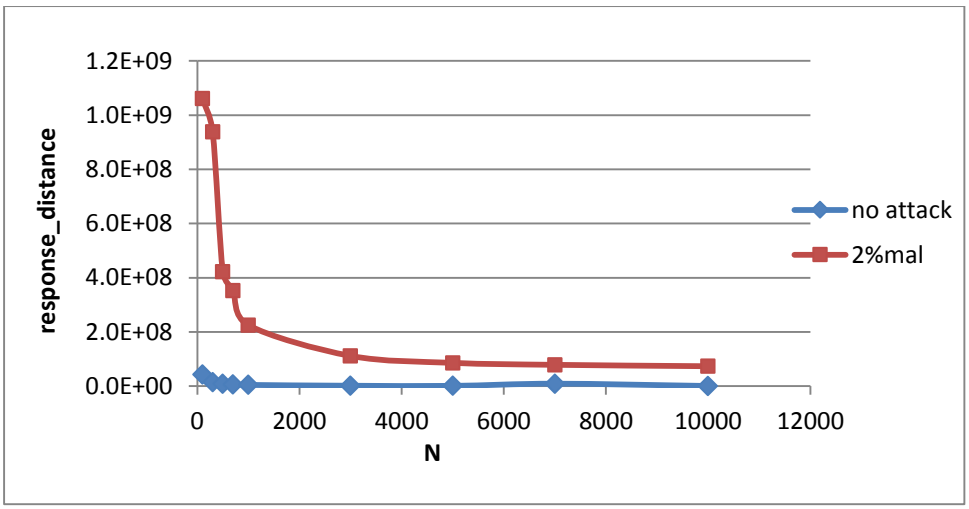

Fig. 15: *Attributo response_distance in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

2) dist_start_finger_compact

Calcolo nel tempo (round r):

$$dist\_start\_finger\_compact(r) = \frac{1}{W}\sum_{k=r-W+1}^{r}\left(\frac{1}{N_f(k)}\sum_{i=1}^{N_f(k)}(finger(k,i) - start(k,i))\right) mod\ 2^m$$

Valore medio (senza attacco):

$$E[dist\_start\_finger\_compact] = \frac{2^m}{N}$$

I campioni con cui si calcola dist_start_finger_compact sono le distanze fra i valori start (node_key + 2^i con $0 \le i \le m - 1$) e i relativi successor (le finger), i quali sono punti casuali sull'anello. Di conseguenza, in base a quanto osservato in [15], deduciamo che gli attributi dist_start_finger_compact e response_distance sono medie campionarie della stessa variabile casuale, ovvero la distanza fra nodi. Ricordiamo che la media della distanza fra nodi è $2^m/N$. Quello che interessa è che dist_start_finger_compact cresce a causa delle entry false poiché queste ultime presentano la differenza (finger – start) mod 2^m come realizzazione di una geometrica di media $2^m/fN$ che quindi è 1/f volte maggiore del caso di entry vera.

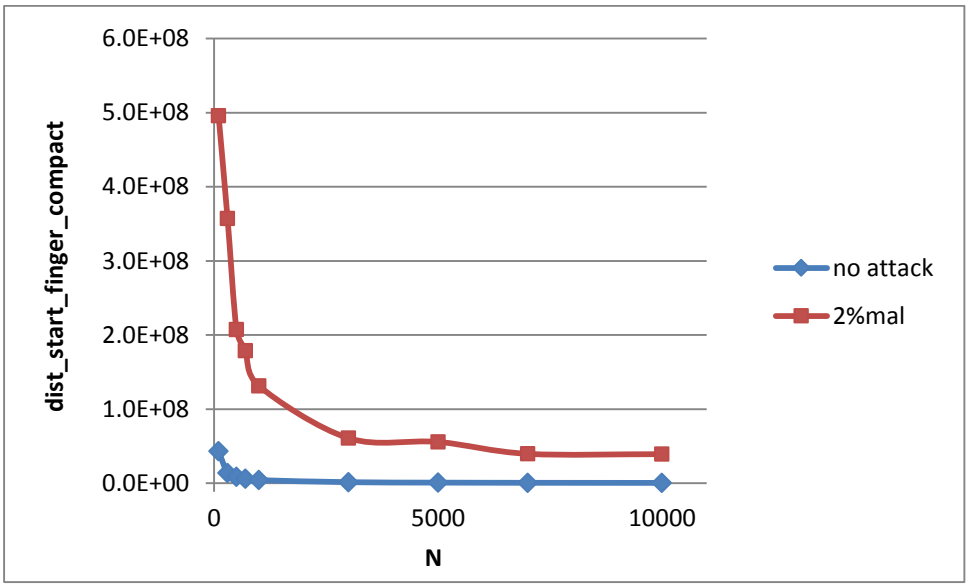

Fig. 16: *Attributo dist_start_finger_compact in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

3) dist_last_finger

Calcolo nel tempo (round r):

$$dist\_last\_finger(r) = \frac{1}{W} \sum_{k=r-W+1}^{r} (last\_finger(k) - current\_node) \quad mod\ 2^m$$

Valore medio (senza attacco):

$$E[dist\_last\_finger] = \frac{2^m}{2} + \frac{2^m}{N}$$

Questo attributo non è altro che la distanza fra il nodo corrente e la sua finger più lontana. Tale grandezza è una realizzazione della distanza fra nodi (si veda il discorso per l'attributo dist_start_finger_compact) a cui si somma la costante $2^m/2$. Il vantaggio qui è che se la finger table è inquinata allora l'ultima finger è maligna con probabilità 1. Dunque questo parametro è ben sensibile alla presenza di maligni.

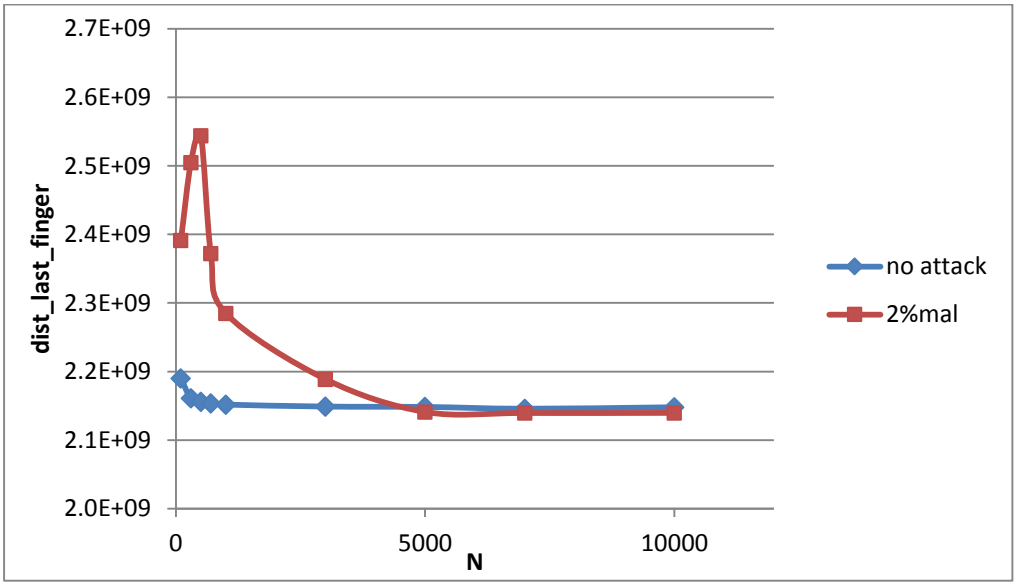

Fig. 17: *Attributo dist_last_finger in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

4) finger_ratio

Calcolo nel tempo (round r):

$$finger\_ratio(r) = \frac{1}{W} \sum_{k=r-W+1}^{r} \left(\frac{1}{N_{mf}(k)} \sum_{i=1}^{N_{mf}(k)} \frac{finger(k,i) - current\_node}{current\_node - previous\_hop(k,i)} \right)$$

Valore medio (senza attacco):

$$finger\_ratio = \frac{1}{2}$$

Un messaggio che giunge ad un nodo onesto deve aver compiuto, nell'ultimo hop, una distanza secondo le regole del protocollo, Infatti i nodi maligni non instradano verso nodi onesti, quindi i messaggi arrivano agli onesti solo da lookup corretti, ossia da onesto a onesto. Pertanto la distanza dell'hop ha valor medio compreso fra $2^1 + 2^m/N$ e $2^{m-1} + 2^m/N$. Se il nodo corrente instrada il messaggio su una finger onesta il salto sarà la metà di quello precedente [2] e quindi fra $2^0 + 2^m/N$ e $2^{m-2} + 2^m/N$. Pertanto, come evidenziato pure dal grafico, questo attributo in condizioni normali ha valore di aspettazione costante a 1/2. Se al contrario la finger è maligna il salto sarà in media più lungo, ossia di valor medio $2^0 + 2^m/fN$ e $2^{m-2} + 2^m/fN$, infatti per definizione $0 \le f \le 1$.

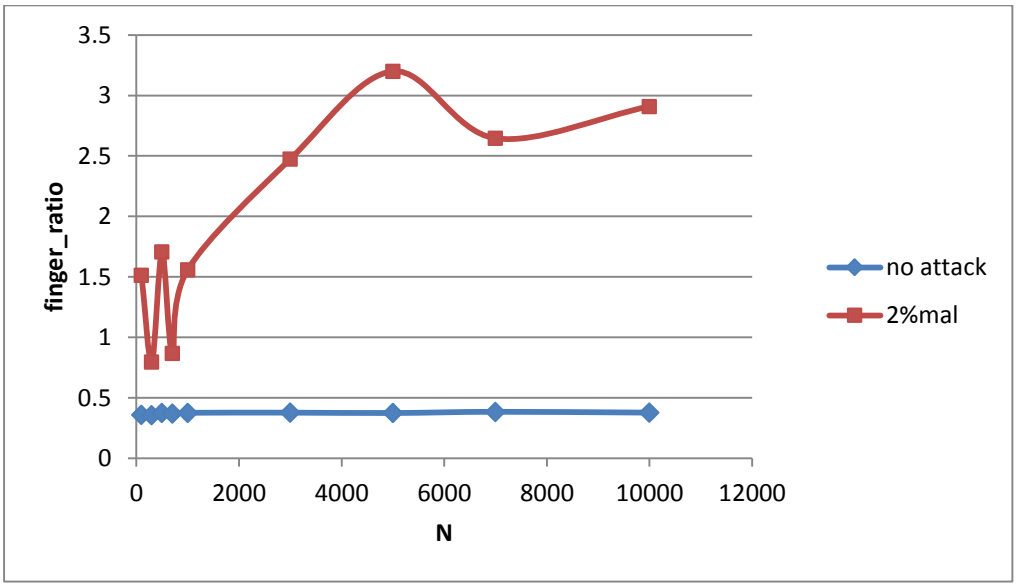

Fig. 18: *Attributo finger_ratio in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

5) ft_length

Calcolo nel tempo (round r):

$$ft\_length(r) = \frac{1}{W} \sum_{k=r-W+1}^{r} finger\_table\_length(k)$$

Valore medio (senza attacco):

$$E[ft\_length] = O(\log_2 N)$$

In [2] è dimostrato che il numero di entry della finger table è $O(\log_2 N)$. La lunghezza della finger table non è costante a m in quanto vengono memorizzate solo le finger diverse fra loro ed il loro numero cresce logaritmicamente con N. Oltre alla debole dipendenza con N, ft_length diminuisce in presenza del subset colluso poiché una finger table inquinata riflette il subset dei nodi collusi, i quali sono una piccola frazione f rispetto al totale dei nodi.

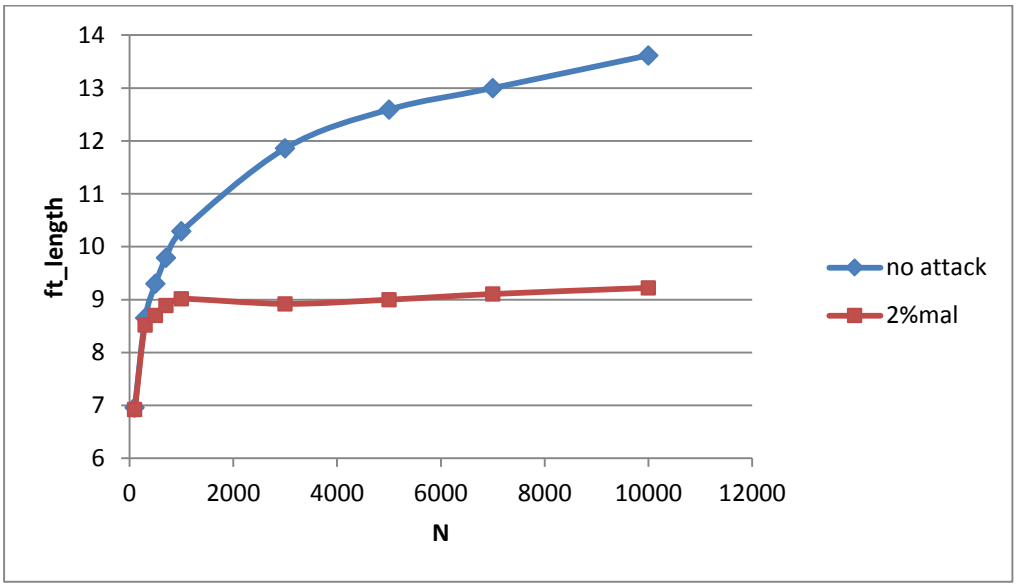

Fig. 19: *Attributo ft_length in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

6) succlist_dist

Calcolo nel tempo (round r):

$$succlist\_dist(r) = \frac{1}{W}\sum_{k=r-W+1}^{r}\left(\frac{1}{N_s}\sum_{i=1}^{N_s}(node(k,i) - node(k,i-1))\right) \mod 2^m$$

Valore medio (senza attacco):

$$E[succlist\_dist] = \frac{2^m}{N}$$

E' la media delle distanze fra i nodi della successor list. Ricordiamo che i campioni usati per calcolare succlist_dist, analogamente a quelli costituenti dist_start_finger_compact e response_distance, sono realizzazioni della stessa variabile casuale, ossia la distanza fra nodi. Pertanto essendo succlist_dist una media campionaria della distanza fra nodi, cresce con la presenza di successori maligni. Essi infatti poiché in generale non sono successori fra loro, contribuiscono con campioni in media 1/f volte superiori i campioni corretti.

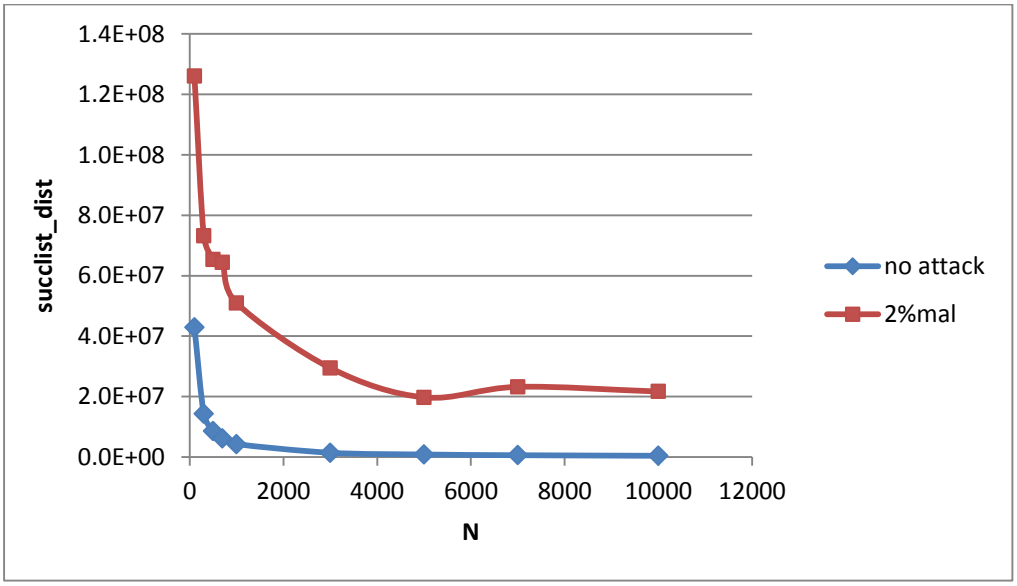

Fig. 20: *Attributo succlist_dist in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

7) hop_count

Calcolo nel tempo (round r):

$$hop\_count(r) = \frac{1}{W} \sum_{k=r-W+1}^{r} \left(\frac{1}{N_{mr}(k)} \sum_{i=1}^{N_{mr}(k)} hop\_count(k,i) \right)$$

Valore medio (senza attacco):

$$E[hop\_count] = \frac{1}{2} \log_2 N$$

L'hop count di un Chord path è logaritmico con N quindi ha una dipendenza sublineare. E' un indicatore di attacco in quanto la cattura dei messaggi dei maligni fa diminuire l'hop count osservato dai nodi onesti.

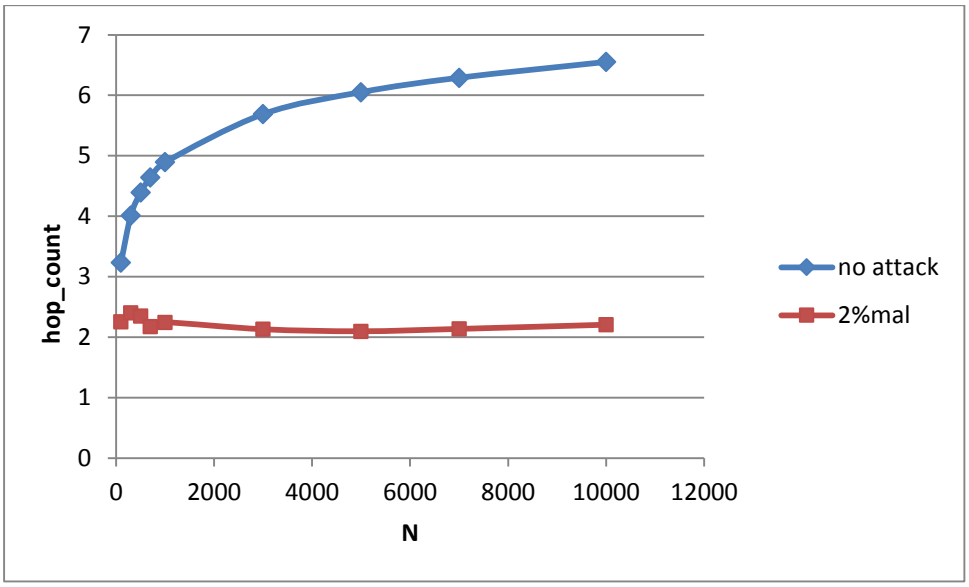

Fig. 21: *Attributo hop_count in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

Gli attributi rimanenti si calcolano, ad ogni round r, come la combinazione degli attributi sopra mostrati:

8) hop_count/log(succlist_dist)

Calcolo nel tempo (round r):

$hop\_count(r)/\log_2(succlist\_dist(r))$

Valore medio (senza attacco):

$$E[\frac{hop\_count}{\log_2(succlist\_dist)}] \approx \frac{E[hop\_count]}{\log_2(E[succlist\_dist])} = \frac{\frac{1}{2}\log_2 N}{\log_2 \frac{2^m}{N}} = \frac{1}{2}\frac{\log_2 N}{m - \log_2 N}$$

L'approssimazione del valore medio è tanto più precisa tanto più sono vere le due condizioni seguenti:

1) Indipendenza statistica fra hop_count e succlist_dist. Questa condizione non si verifica esattamente, tuttavia hop_count varia con N molto più lentamente rispetto a succlist_dist.

2) Minore è la varianza della variabile casuale succlist_dist. Essa è una media campionaria della distanza fra nodi. Essendo quest'ultima una geometrica, tale varianza diminuisce all'aumentare del numero di nodi N nella rete.

In ogni caso, i valori numerici della curva senza attacco confermano la legittimità dell'approssimazione

Questo attributo ha la proprietà di essere molto dipendente dai nodi maligni, infatti con essi il numeratore decresce e il denominatore incrementa. Inoltre l'attributo ha una debole dipendenza (sublineare) con N.

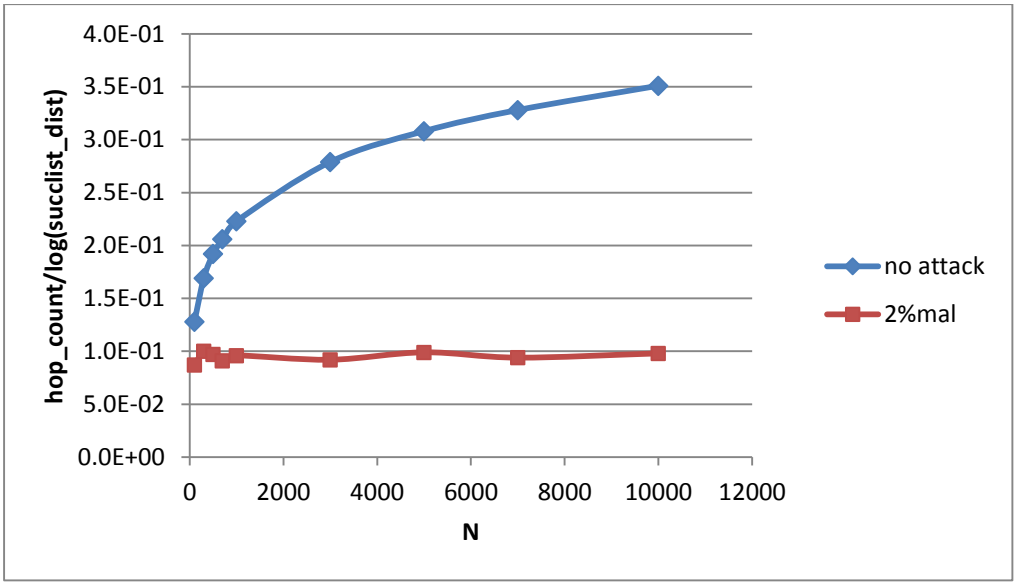

Fig. 22: *Attributo hop_count/log(succlist_dist) in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

9) (2^2·hop_count)·dist_start_finger_compact

Calcolo nel tempo (round r):

$$\left(2^{2\cdot hop\_count(r)}\right)\cdot dist\_start\_finger\_compact(r)$$

Valore medio (senza attacco):

$$E\left[\left(2^{2\cdot hop\_count}\right)\cdot dist\_start\_finger\_compact\right] \approx$$
$$\approx \left(2^{2\cdot E[hop\_count]}\right)\cdot E[dist\_start\_finger\_compact] =$$
$$= 2^{2\cdot\frac{1}{2}\log_2 N}\cdot\frac{2^m}{N} =$$
$$= N\cdot\frac{2^m}{N} =$$
$$= 2^m$$

Come nel caso precedente, la curva senza attacco testimonia che l'approssimazione della formula sopra in questo caso è lecita.
Questo attributo combina i due attributi hop_count e dist_start_finger_compact in modo da essere indipendente da N. In assenza di nodi maligni, dal grafico si nota che l'attributo rimane sempre attorno a 2^m, infatti con m = 32 si ha $2^{32} = 4.29\cdot10^9$.

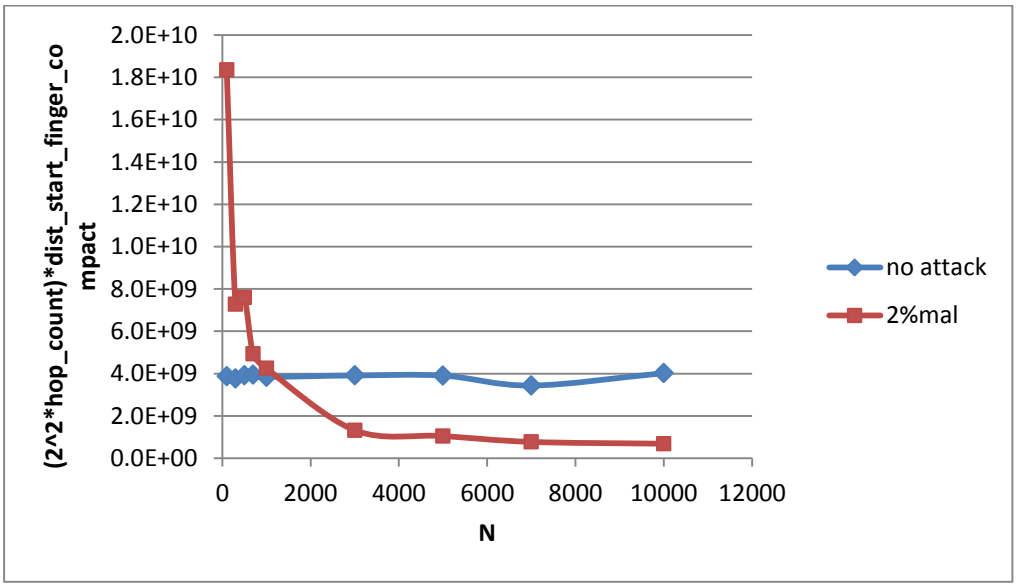

Fig. 23: *Attributo (2^2·hop_count)·dist_start_finger_compact in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

10) (2^2·hop_count)·succlist_dist

Calcolo nel tempo (round r):

$$\left(2^{2 \cdot hop\_count(r)}\right) \cdot succlist\_dist(r)$$

Valore medio (senza attacco):

$$E\left[\left(2^{2 \cdot hop\_count(r)}\right) \cdot succlist\_dist(r)\right] \approx$$
$$\approx \left(2^{2 \cdot E[hop\_count]}\right) \cdot E[succlist\_dist] =$$
$$= 2^{2 \cdot \frac{1}{2} \log_2 N} \cdot \frac{2^m}{N}$$
$$= N \cdot \frac{2^m}{N}$$
$$= 2^m$$

Vale quanto detto per l'attributo precedente, infatti succlist_dist è la media campionaria della distanza fra nodi, come lo è dist_start_finger_compact.

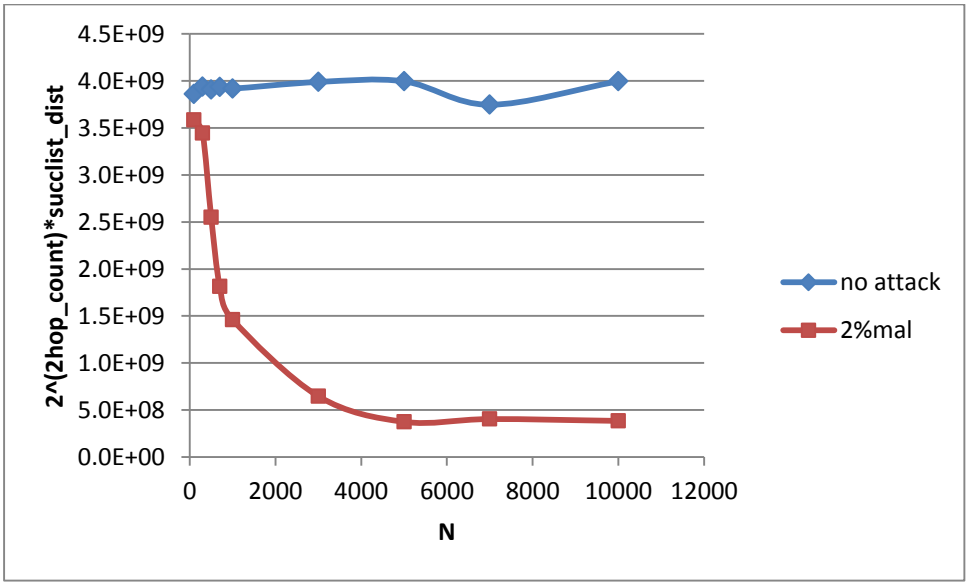

Fig. 24: *Attributo (2^2·hop_count)·succlist_dist in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

11) dist_start_finger_compact/hop_count

Calcolo nel tempo (round r):

$$dist\_start\_finger\_compact(r)/hop\_count(r)$$

Valore medio (senza attacco):

$$\frac{dist\_start\_finger\_compact}{hop\_count} \approx \frac{E[dist\_start\_finger\_compact]}{E[hop\_count]} = = \frac{\frac{2^m}{N}}{\frac{1}{2}log_2\, N} = = \frac{2 \cdot 2^m}{N\, log_2\, N}$$

L'attributo presenta una spiccata dipendenza con i nodi maligni, infatti come già spiegato, in presenza di attacco il numeratore cresce mentre il denominatore decresce.

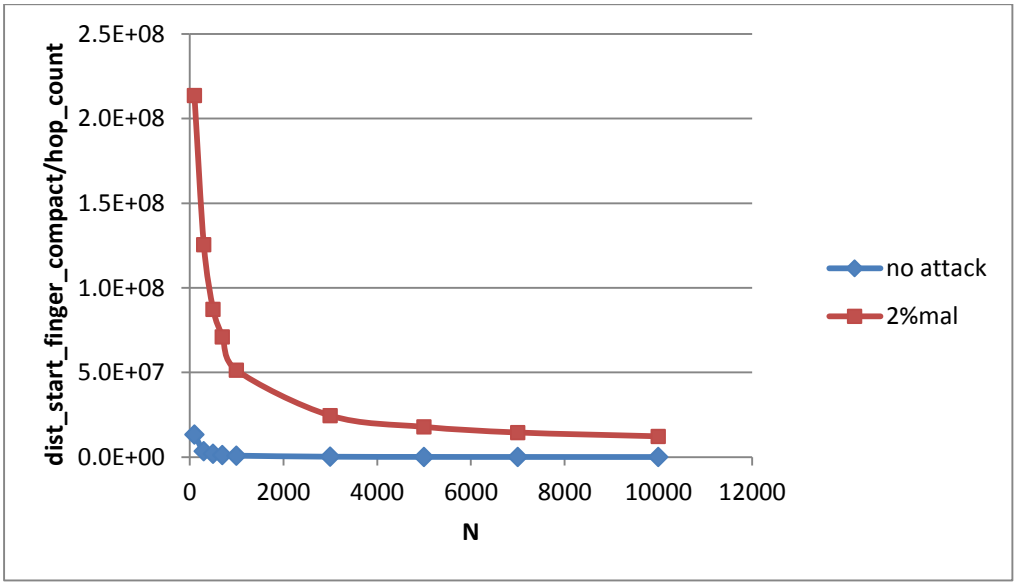

Fig. 25: *Attributo dist_start_finger_compact/hop_count in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

12) 2·hop_count+log(dist_start_finger_compact)

Calcolo nel tempo (round r):

$$2 \cdot hop\_count(r) + \log_2(dist\_start\_finger\_compact(r))$$

Valore medio (senza attacco):

$$E[2 \cdot hop\_count + \log_2(dist\_start\_finger\_compact)] \approx$$
$$\approx 2 \cdot E[hop\_count] + \log_2(E[dist\_start\_finger\_compact]) =$$
$$= 2 \cdot \frac{1}{2}\log_2 N + log_2\left(\frac{2^m}{N}\right) =$$
$$= \log_2 N + m - \log_2 N =$$
$$= m$$

In [2] è dimostrato che il valore medio di hop_count è $(1/2)\log_2 N$ invece succlist_dist come già affermato, è una media campionaria della distanza fra nodi, la cui media è $2^m/N$. Dall'espressione sopra si conclude che il valore medio dell'attributo è pari al valore della lunghezza in bit della chiave m, perciò è indipendente da N.

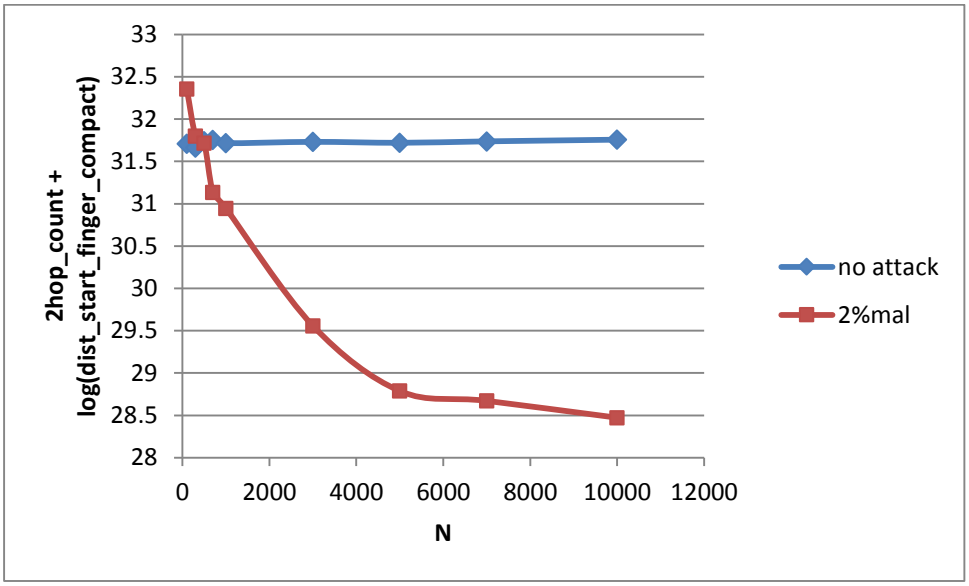

Fig. 26: *Attributo 2·hop_count+log(dist_start_finger_compact) in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

13) 2·hop_count + log(succlist_dist)

Calcolo nel tempo (round r):

$$2 \cdot hop\_count(r) + \log_2(succlist\_dist(r))$$

Valore medio (senza attacco):

$$E[2 \cdot hop\_count + \log_2(succlist\_dist)] \approx$$
$$\approx 2 \cdot E[hop\_count] + \log_2(E[succlist\_dist]) = 2 \cdot \frac{1}{2}\log_2 N + log_2\left(\frac{2^m}{N}\right) =$$
$$= \log_2 N + m - \log_2 N =$$
$$= m$$

Valgono le considerazioni dell'attributo precedente poiché succlist_dist si comporta come dist_start_finger_compact. Anche il grafico ne dà conferma.

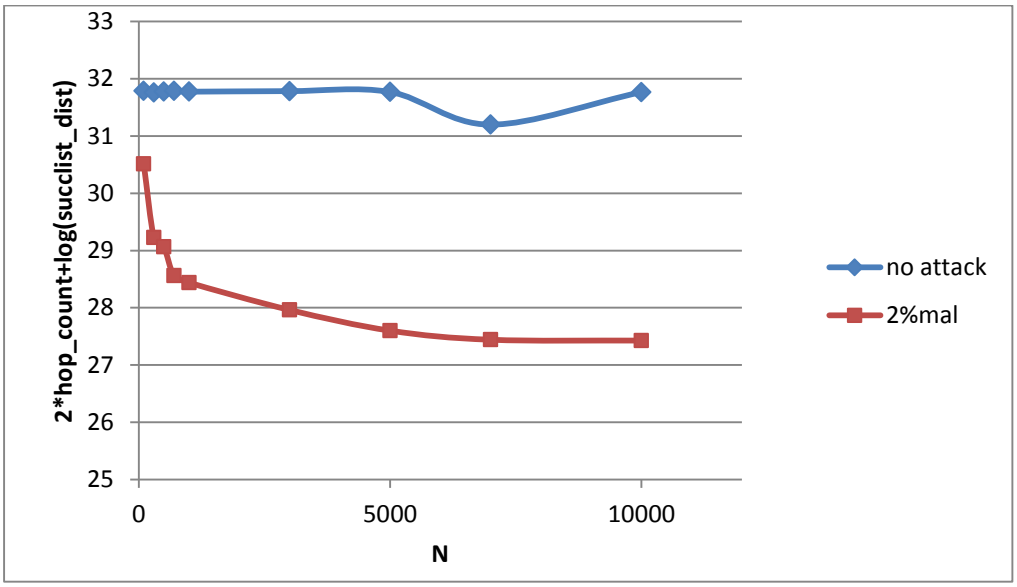

Fig. 27: *Attributo 2·hop_count + log(succlist_dist) in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

14) succlist_dist·dist_start_finger_compact

Calcolo nel tempo (round r):

$$succlist\_dist(r) \cdot dist\_start\_finger\_compact(r)$$

Valore medio (senza attacco):

$$E[succlist\_dist \cdot dist\_start\_finger\_compact] \approx$$
$$\approx E[succlist\_dist] \cdot E[dist\_start\_finger\_compact] =$$

$$= \frac{2^m}{N} \cdot \frac{2^m}{N} =$$

$$= \frac{2^{2m}}{N^2}$$

Questo attributo ha una forte dipendenza con l'attacco, infatti i due attributi succlist_dist e dist_start_finger_compact sono più elevate in presenza di maligni.

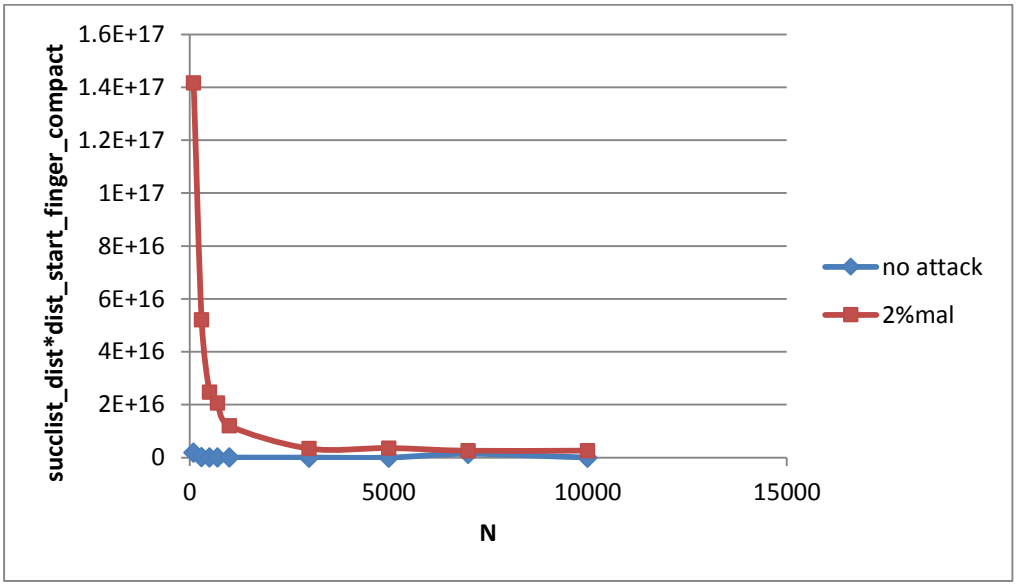

Fig. 28: *Attributo succlist_dist·dist_start_finger_compact in funzione del numero di nodi N e della frazione f di nodi maligni (f = 0 e f = 0.02).*

E' essenziale osservare che l'attributo ideale darebbe due insiemi di punti giacenti su una retta parallela all'asse N, ossia punti incorrelati con N e non dispersi. Anche se questo non accade, l'importante è che le curve non si incrocino per effetto della dipendenza con N. Come si osservato nei grafici, la maggioranza degli attributi, per il range di N considerato (da 100 a 10000), soddisfano quest'ultima caratteristica.

Adesso ci chiediamo quali attributi del training set sono effettivamente utili, ossia quali di essi sono inclusi nell'albero. L'algoritmo C4.5 effettua già da sé una selezione degli attributi migliori, infatti durante la costruzione dell'albero, ad ogni nodo non foglia, il C4.5 individua il miglior attributo, sempre a partire dal set degli attributi iniziale. Esistono diversi indici per quantificare la capacità discriminante di un attributo: information gain, gain ratio, gini index, etc. Nel caso del C4.5 viene usato il gain ratio. In pratica, ad ogni nodo non foglia, vengono ordinate le istanze del training set in senso crescente rispetto ad ogni attributo, e poi per ogni valore dell'attributo considerato, viene calcolato il gain ratio. Per ogni attributo viene poi

memorizzato il gain ratio maggiore, cioè quello relativo alla soglia che consente di discriminare le classi con il minimo errore di classificazione. La coppia (attributo, soglia) scelta al nodo non foglia in questione è quella che, nel nodo in questione, possiede il maggior gain ratio. Questa operazione di selezione della migliore combinazione (attributo, soglia) è ripetuta ricorsivamente ad ogni nodo non foglia, percui la selezione degli attributi più utili è intrinseca nella costruzione dell'albero C4.5. Ovviamente qui per attributi migliori o più utili intendiamo quelli migliori secondo il criterio del gain ratio (algoritmo C4.5). Per completezza abbiamo anche isolato gli attributi migliori attraverso la lunga serie di algoritmi ad hoc di selezione degli attributi offerti da Weka (si veda Explorer → Select attributes). Tuttavia, spesso accade che riducendo gli attributi del training set, l'albero aumenta velocemente di dimensione. Per questo motivo non abbiamo effettuato la riduzione dello spazio degli attributi, se non nell'intento di rimuovere quelli polinomialmente dipendenti con N, come vedremo. Come invece abbiamo notato, gli attributi polinomiali in N sono sostanzialmente delle iperboli in N di grado mai superiore a 2, perciò neanche troppo dipendenti con N. Ricordiamo che la dimensione dell'albero è il numero di nodi, mentre il numero di costrutti if / else che codificano lo splitting di ogni nodo è il numero di nodi escluse le foglie. E' chiaro quindi che si vuole avere il minor numero possibile di nodi non foglia al fine di avere un modello che simultaneamente richiede meno computazione ed è codificato in modo più compatto.

5.3 Risultati

Abbiamo addestrato l'albero attraverso tre set di training diversi che per semplicità indichiamo con L, H e LH. La differenza fra di essi è il diverso range di N coperto, come si può notare nelle seguenti tre tabelle. In esse facciamo vedere la composizione dei tre training set: ogni riga della tabella è uno scenario di simulazione in cui la prima colonna riporta il numero di nodi dell'overlay, la seconda il numero di istanze di training (righe nel file di training set), e f è la percentuale di nodi maligni.

1) Training L

N	No. instances	f
100	1000	0
100	1000	[0.01,0.05]
500	1500	0
500	1500	[0.01,0.05]
1000	2000	0
1000	2000	[0.01,0.05]
Avg = 766	Tot = 9000	

Tab. 1: *Composizione del training set L.*

2) Training H

N	No. instances	f
1000	1000	0
1000	1000	0,02
5000	5000	0
5000	5000	0,02
10000	10000	0
10000	10000	0,02
Avg = 7875	Tot = 32000	

Tab. 2: *Composizione del training set H.*

3) Training LH

N	No. instances	f
100	2000	0
100	2000	[0.01,0.05]
400	800	0
400	800	[0.01,0.05]
500	5000	0
500	5000	[0.01,0.05]
1000	3000	0
1000	3000	[0.01,0.05]
2000	4000	0
2000	4000	[0.01,0.05]
5000	5000	0
5000	5000	0,02
10000	5000	0

10000	5000	0,02
Avg = 3588	Tot = 49600	

Tab. 3: *Composizione del training set LH.*

E' da sottolineare inoltre che l'insieme degli attributi considerati da un albero decisionale è compreso o coincidente con l'insieme degli attributi del training set impiegato per generarlo. Per ognuno dei tre training set (L, H e LH), abbiamo generato mediante l'algoritmo C4.5 due alberi decisionali differenti in base agli attributi presenti nei training set, attraverso una selezione manuale degli attributi. Tale distinzione degli attributi usati nel training set consiste nei seguenti due set di attributi:

1) Complete

E' il set completo di attributi (elencati in precedenza).

2) Indip_N

Qui si considerano solo gli attributi i quali, almeno dal punto di vista analitico, sono sublinearmente dipendenti o non dipendenti da N. Ricordiamo tali attributi:

- hop_count
- hop_count/log(dist_start_finger)
- hop_count/log(dist_start finger_compact)
- (2^2hop_count)·dist_start_finger_compact
- finger_ratio
- 2hop_count+log(succlist_dist)
- 2hop_count+log(dist_start_finger_compact)
- hop_count/log(succlist_dist)
- 2^(2·hop_count)·succlist_dist
- ft_length

Combinando i tre training L, H, LH e i due set di attributi, in teoria ci sono 6 alberi possibili. Tuttavia consideriamo qui i tre più significativi:

1) Complete (L)
2) Complete (H)
3) Indip_N (LH)

Vediamo ora perché abbiamo combinato i training L, H e LH con certi attributi. I primi due alberi, poiché hanno un training set che copre un intervallo di N abbastanza limitato, possono permettersi di impiegare anche gli attributi polinomiali in N. Per tale motivo, nell'atto della loro costruzione tramite l'algoritmo C4.5, abbiamo lasciato il set completo di attributi, dunque anche quelli polinomiali in N. Il terzo albero, poiché impiega un training set più ampio rispetto ad N, è stato limitato ai soli attributi sublineari in N. Ciò significa che nel momento della generazione dell'albero, abbiamo manualmente rimosso gli attributi che dipendono i modo polinomiale con N.
Nelle pagine seguenti presentiamo gli attributi considerati da ognuno dei tre alberi e le tabelle che riportano la percentuale di classificazioni corrette. Nei commenti seguenti, per accuratezza soddisfacente, intendiamo una soglia minima del 90% di classificazioni corrette, ossia una probabilità di classificazione corretta almeno di 0.9. Per quanto riguarda il sistema di tracciamento degli attributi, abbiamo fissato i seguenti valori del sistema a sliding window: T_r = 200 s e W = 10 round. Ogni colonna delle tabelle che mostrano la qualità di questo sistema di detection è uno scenario di simulazione, indicato con la seguente notazione: N_%nodimaligni dove N, ricordiamo, è il numero di nodi dell'overlay.

1) <u>Complete L</u>

Attributi nell'albero:

- response_distance

Numero attributi = 1
Numero nodi = 3
Numero foglie = 2

Percentuali classificazioni corrette:

Complete (L)	**% Correct Classifications**							
time [s]	**300_0**	**300_2**	**700_0**	**700_2**	**3000_0**	**3000_2**	**7000_0**	**7000_2**
1000	100	100	100	100	100	99	100	20
2000	100	100	100	100	100	99	100	18
3000	100	100	100	100	100	99	100	17
4000	100	100	100	100	100	99	100	14
5000	100	100	100	100	100	99	100	13

Tab. 4: *Risultati dell'albero Complete L nei diversi scenari in cui variano il numero di nodi (N = 300, 700, 3000, 7000) e la frazione di nodi maligni (f = 0 e f = 0.02).*

Essendo l'attributo response_distance fortemente dipendente da N, e dato che il training set L copre scenari da 100 a 1000 nodi, questo albero dà prestazioni carenti per lo scenario 7000_02. In tutti gli altri casi è invece caratterizzato da un'ottima accuratezza. Inoltre, essendo un albero banale, ossia un semplice test a soglia, è il modello meno costoso in termini di overhead.

2) Complete H

Attributi nell'albero:

- response_distance

Numero attributi = 1
Numero nodi = 3
Numero foglie = 2

Percentuali classificazioni corrette:

Complete (H)	**% Correct Classifications**							
time [s]	**300_0**	**300_2**	**700_0**	**700_2**	**3000_0**	**3000_2**	**7000_0**	**7000_2**
1000	0	100	1	100	100	100	100	100
2000	0	100	1	100	100	100	100	100
3000	0	100	1	100	100	100	100	100
4000	0	100	2	100	100	100	100	100
5000	0	100	2	100	100	100	100	100

Tab. 5: *Risultati dell'albero Complete H nei diversi scenari in cui variano il numero di nodi (N = 300, 700, 3000, 7000) e la frazione di nodi maligni (f = 0 e f = 0.02).*

Essendo l'attributo Response Distance fortemente dipendente da N, e dato che il training set H copre scenari da 1000 a 10000 nodi, questo albero dà prestazioni carenti per gli scenari 300_00 e 700_00. In tutti gli altri casi è invece caratterizzato da un'ottima accuratezza. Anche qui sull'overhead vale quanto affermato nel caso precedente.

3) Indip_N LH

Attributi nell'albero:

- hop_count
- hop_count/log(dist_start_finger_compact)
- (2^2·hop_count)·dist_start_finger_compact
- finger_ratio
- 2hop_count+log(succlist_dist)
- 2hop_count+log(dist_start_finger_compact)
- hop_count/log(succlist_dist)
- ft_length

Numero attributi = 8
Numero nodi = 145
Numero foglie = 73

Percentuali classificazioni corrette:

Indip_N (LH)	**% Correct Classifications**							
time [s]	**300_0**	**300_2**	**700_0**	**700_2**	**3000_0**	**3000_2**	**7000_0**	**7000_2**
1000	90	98	96	99	98	99	98	99
2000	90	98	96	98	98	99	98	99
3000	91	97	97	98	98	99	98	99
4000	92	97	96	99	98	99	97	99
5000	92	97	95	99	97	99	97	99

Tab. 6: *Risultati dell'albero Indip_N LH nei diversi scenari in cui variano il numero di nodi (N = 300, 700, 3000, 7000) e la frazione di nodi maligni (f = 0 e f = 0.02).*

La tabella sopra evidenzia un'accuratezza soddisfacente su tutto il range di N, confermando l'effettiva robustezza rispetto ad N del subset di attributi Indip_N. Tuttavia, non si ha mai la massima accuratezza che invece era possibile avere con il set completo degli attributi.

E' interessante osservare come gli alberi ottenuti con il set completo degli attributi e con training limitato in N (L,H), di fatto non sono alberi, bensì dei semplici test a soglia. Ciò fa dedurre che, quando il range di N sul training set è limitato, response_distance è sufficiente per discriminare perfettamente le istanze. In ogni caso, il fatto che se si utlizzano solo gli attributi Indip_N non si raggiunge la stessa accuratezza di response_distance (non appartentente a Indip_N), porta a dedurre che siamo di fronte ad un trade-off: minore è la dipendenza con N degli attributi e minore è la loro dipendenza con l'attacco. Questo perché l'effetto della presenza dei nodi collusi è lo stesso effetto di avere un rete con N inferiore. Quindi se si indebolisce la dipendenza con N, lo si fa altrettanto con la dipendenza rispetto all'attacco. Di conseguenza, a patto di conoscere il range di N nell'applicazione in questione, il test su response_distance dà l'accuratezza migliore. Inoltre ci sono altri due punti essenziali a favore dell'attributo response_distance. Il primo è che è meno affetto da eventuali difese rispetto agli altri: siccome le difese in generale tendono a limitare l'inquinamento della successor list e della finger table, gli attributi basati sulle proprietà della successor list e della finger table, risentono dell'effetto delle difese inducendo a segnalare l'assenza di attacco. Ciò oltretutto induce un comportamento oscillante di identificazione dell'attacco e quindi di attivazione e disattivazione delle difese. Questi sospetti sono stati pure confermati dalle simulazioni che per compattezza non abbiamo qui riportato. Un modo per aggirare questo inconveniente è quello di usare apposite informazioni di routing indifese, cioè alle quali non si applicano le contromisure e che hanno l'unico scopo di riflettere lo stato

della rete. Il secondo vantaggio di response_distance è che non dipende dalla distribuzione del traffico sull'anello generato dagli altri nodi. Al contrario, l'attributo hop_count osservato da un certo nodo situato nell'anello in un punto di chiave n, dipende dal traffico passante per n. Questo fattore di disturbo, ovvero la distribuzione del traffico sull'anello, fa pensare che hop_count non sia flessibile rispetto al traffico. Crediamo infatti che l'algoritmo C4.5 abbia incluso nell'albero l'attributo hop_count e combinazioni di attributi contenenti hop_count, solo perchè tutti gli scenari che compongono i training set si riferiscono al caso ideale, ossia quello in cui tutti i nodi generano traffico uniforme sull'anello. Ciò implica che hop_count sia coerente da nodo a nodo perchè tutti i nodi vedono pressoché lo stesso traffico. Lo stesso ragionamento si può fare anche per l'attributo finger_ratio, dato che anche'esso nasce dai messaggi osservati. Al contrario response_distance non dipende dalla posizione del nodo nell'anello, né tantomeno dal traffico transitante. Gli unici limiti di response_distance sono due: 1) Come tutti gli attributi, occorre avere una sufficiente confidenza statistica. 2) Dipende dalla distribuzione sull'anello del traffico che sorge nel nodo. Entrambi questi problemi dipendono dal comportamento dell'applicazione. Sappiamo che le risposte possono essere gli overlay ack e/o le fix finger response. Le ultime sono stimolate dalle fix finger call, le quali coprono sempre metà anello, quindi costituiscono un insieme di campioni abbastanza rappresentativo circa la situazione dell'overlay. L'origine degli overlay ack invece dipende dalle chiavi di destinazione dei messaggi sorti nel nodo ed esse sono determinate dall'applicazione. Per rendere l'origine degli overlay ack indipendenti dall'applicazione e renderli rappresentativi, ossia provenienti da tutto l'anello, si potrebbero trasmettere periodicamente messaggi di probe uniformi sull'anello in modo da stimolare le rispettive risposte. Nelle nostre simulazioni non ci siamo posti tuttavia questo problema poiché ogni nodo genera già traffico uniforme sull'anello e con sufficiente frequenza. Infine, è importante evidenziare che il test response_distance ha il minimo overhead possibile per un albero.

Da ultimo, alcune note sulla possibile implementazione. Poiché la raccolta degli attributi si riferisce ad eventi di competenza di Chord (per esempio l'arrivo di un messaggio Chord) e a strutture dati di Chord (finger table e successor list), il sistema di misura degli attributi dev'essere integrato nello stesso software che implementa Chord. Al contrario il sistema a sliding window dell'aggiornamento degli attributi e classificazione binaria, poiché usano i valori già raccolti, non devono necessariamente essere integrati nel codice di Chord. Pertanto essi possono essere un software indipendente impiegato da Chord stesso come libreria. Circa la funzione di classificazione, essendo un albero, può essere codificata come un insieme di regole if / else.

6
CONTROMISURE ALL'ATTACCO

6.1 Premessa

Di seguito elenchiamo i significati e le assunzioni che rimangono implicite nei paragrafi a venire, in modo da evitare ripetizioni:

- h = hop count di un path totale di un messaggio
- E[h] = hop count medio di tutti i messaggi "utili" cioè terminati su nodi onesti (M è il numero totale di tali messaggi nella simulazione).

$$E[h] = \frac{1}{M}\sum_{i=1}^{M} h(i)$$

- E[h_{rel}] = hop count relativo medio di tutti i messaggi "utili" cioè terminati su nodi onesti

$$normalized\_distance(i) = \frac{dest\_node(i) - source\_node(i)}{2^m}$$

$$E[h_{rel}] = \frac{1}{M}\sum_{i=1}^{M} \frac{h(i)}{normalized\_distance(i)}$$

- Socket_size = 4 byte (IPv4 + port) o 10 byte (IPv6 + port)

6.2 Introduzione

Ogni proposta di difesa deve rispondere a tre requisiti:

- Efficacia (in termini di riduzione di messaggi catturati)
- Scalabilità con il numero di nodi
- Non deve presentare vulnerabilità

Tutte le proposte di difese elaborate in questa tesi si fondano su un principio: una difesa è un'azione che normalmente il protocollo non esegue ed è eseguita in modo da ottenere uno stato e/o un comportamento che aderisca il più possibile alle regole del protocollo.
In ogni overlay distribuito, la difesa ideale contro Eclipse consente di ridurre l'attacco Eclipse al decisamente meno dannoso attacco Sybil. Se poi la difesa fosse in grado di classificare come maligne tutte le risposte maligne, allora tale difesa non solo renderebbe vano l'attacco Eclipse ma anche l'attacco Sybil perchè ogni nodo sarebbe in grado di escludere a priori dal proprio neighborhood le entry maligne.
In questa tesi ci siamo principalmente focalizzati sulle difese distribuite o decentralizzate. In altre parole, la difesa stessa è preferibile che sia peer to peer, esattamente come lo è Chord. Questo significa che la difesa, identicamente al protocollo Chord, non deve avvalersi di server / supernodi esterni che supervisionano la rete. L'intento è perciò quello di preservare la caratteristica peculiare di Chord, ovvero la sua natura completamente distribuita, dove tutti i nodi hanno il medesimo ruolo a livello overlay. Solo così si mantiene la flessibilità e la scalabilità che distingue il paradigma peer to peer decentralizzato strutturato da uno gerarchico.
E' essenziale riconoscere che l'ipotesi peer to peer decentralizzata della difesa è un vincolo fortemente stringente e che quindi rende difficile ottenere un sistema di difesa semplice ed efficace. La conseguenza del fatto che la difesa dev'essere peer to peer distribuita (come lo è Chord) è che, analogamente a Chord, non può usufruire di tutte le informazioni necessarie ad avere la difesa ottimale ma deve accontentarsi delle sole informazioni già fornite da Chord o comunque può recuperare informazioni aggiuntive in misura limitata.
Perchè l'attacco Eclipse è difficile da debellare in Chord? Il motivo è semplice: perchè in Chord quando un nodo entra nella rete, possiede la minima informazione possibile sugli altri nodi presenti nella rete. Infatti quando il nodo entra nella rete conosce solo il bootstrap node dal quale ottiene il successor. Tutta la restante informazione dell'overlay viene guadagnata prima dal successor e poi ricorsivamente dai nodi man mano conosciuti. Questo implica che

se anche solo una entry maligna entra nel neighborhood del nodo, questa entry maligna, puntando esclusivamente ad altri nodi maligni, incrementa la percentuale maligna del neighborhood e così via.
Se invece ogni nodo, al momento di entrare alla rete, conoscesse già la chiave di tutti gli N nodi presenti nella rete, allora l'attacco Eclipse non riuscirebbe ad inquinare attivamente i nodi onesti e la sua efficacia sarebbe ridotta al livello del Sybil. In questo caso l'attacco Eclipse non è ancora sconfitto finché non si identificano i nodi maligni e quindi si evita di instradare messaggi verso di essi. I nodi maligni infatti, anche se non riescono a inquinare attivamente i nodi onesti, possono comunque catturare i messaggi (Sybil). Nel caso di Chord vedremo come molte difese proattive si basano sull'ottenimento di una maggiore quantità di informazioni di routing rispetto a quelle che si ottengono secondo le normali funzioni Chord. Ovviamente, in nessuna difesa un nodo non pretende mai di avere la visione globale della rete poiché questa non sarebbe una soluzione scalabile.
La difesa ottima contro l'attacco Eclipse è quella che presso ogni nodo è in grado di identificare tutti i nodi maligni della rete. Così facendo ogni nodo onesto non invierebbe nessuna chiave a nodi che cadono nell'insieme dei nodi maligni e pertanto la percentuale di chiavi catturate dai maligni sarebbe nulla e quindi anche l'attacco Sybil sarebbe completamente neutralizzato. Una tale difesa, escludendo i nodi maligni dal sistema, quindi non solo annulla l'attacco Sybil ma pure l'attacco Eclipse dato che l'attacco Eclipse non è altro che un potenziamento funzionale dell'attacco Sybil. Questa è la soluzione ottima dal punto di vista teorico ma non dal punto di vista pratico dato che la classificazione dei nodi in onesti o maligni non è mai esatta e quindi è affetta da falsi positivi e negativi. Il problema è che non è banale, in generale, classificare in modo accurato la malignità/onestà di un nodo. Il compito di classificare se un nodo è onesto o maligno non è banale poiché si deve basare sull'osservazione di una serie di comportamenti dei nodi e ciò non è garantito che dia buoni risultati per due motivi: 1) Si tratta di misure statistiche nel tempo, in generale soggette a rumore statistico e che possono richiedere un tempo non indifferente prima di dare risultati statisticamente significativi. 2) Queste misure non dipendono solo dall'onesta/malignità dei nodi ma anche da altri fattori come ad esempio la frequenza con cui i messaggi sono generati, la distribuzione statistica delle chiavi, ecc. 3) Come avevamo visto sull'attacco Eclipse, i nodi maligni aderiscono il più possibile al comportamento originale in modo da rendere arduo il loro riconoscimento.
In ogni caso la soluzione ottima, cioè contro l'attacco Sybil e quindi anche Eclipse, può essere pensata nel seguente modo: si deve disporre di un insieme di supernodi che tengono traccia dell'ingresso e uscita di tutti i nodi nella rete e che li classifica come onesti o maligni: a quel punto ogni nodo onesto può ricevere la lista dei nodi maligni dal supernodo. In questa

soluzione ottimale, la presenza del set di supernodi avente la visione totale della rete, equivale ad avere un sistema di controllo overlay, per cui non è in linea con il modello overlay puramente peer to peer nel quale siamo per ipotesi. Questa soluzione dei supernodi è sì fattibile ma applicabile solo a reti con un numero di nodi abbastanza modesto poiché, in quanto architettura centralizzata, è carente sotto l'aspetto della resilienza e della scalabilità col numero dei nodi. Il primo aspetto è dovuto al fatto che i supernodi e la rete fisica (rete underlay) che li collegano rappresentano un collo di bottiglia per la dipendibilità del sistema e il secondo risiede nel fatto che diventa arduo gestire milioni di nodi in modo centralizzato, sia per i supernodi stessi sia soprattutto per la rete fisica con cui si collegano.
Si potrebbe allora pensare, per questa soluzione ottima, alla rispettiva alternativa completamente distribuita, cioè senza supernodi. L'idea è che ogni nodo onesto viene a conoscenza di tutti i maligni per mezzo del sistema di propagazione tra i nodi onesti dei nodi maligni individuati. Questa soluzione tuttavia non scala bene con il numero di nodi ed è soprattutto una vulnerabilità in sé perché può essere sfruttata dai nodi maligni.

In questo lavoro di tesi è emerso che non esiste un singolo algoritmo di difesa in Chord contro l'attacco Eclipse. Il motivo è che l'attacco Eclipse in Chord agisce su diversi aspetti del funzionamento di Chord. Piuttosto, abbiamo proposto una moltitudine di algoritmi di difesa, ognuno di essi perfettamente integrabile nel software Chord. Tali algoritmi di difesa possono essere combinati fra loro in modo arbitrario purché siano mutualmente compatibili. Ovviamente più difese si aggiungono, cioè più è ampio il set di difesa, maggiore è superiore l'overhead computazionale, di memoria e di traffico.

6.3 Proposte

Prima di vedere le singole proposte di contromisura all'attacco Eclipse in Chord, presentiamo qui due primitive che abbiamo introdotto:

- Distance Test
- Nodelist

La prima è una funzione mentre le seconda è una struttura dati. Molte difese proposte in questa tesi usufruiscono di queste due primitive.

Distance Test

Come vedremo molte delle nostre difese distribuite si avvalgono su una classificazione binaria attraverso un test a soglia. Questa tecnica la chiamiamo Distance Test perchè è un test per classificare se una risposta è vera o falsa. Per risposta intendiamo un overlay ack o una fix finger response e si legga il par. 4.1 per maggiori informazioni. Come si vedrà il risultato della classificazione della risposta serve a prendere diverse tipologie di contromisure a seconda della difesa in questione. L'idea di Distance Test è quella di sfruttare la risposta come feedback per reagire in base all'esito dedotto da esso. Il criterio per il quale un nodo onesto classifica la risposta come vera o falsa è vedere quanto verosimilmente tale risposta è stata originata dal successor del messaggio diretto che l'ha causata o no. Per messaggio diretto intendiamo un messaggio applicativo se intendiamo come risposta l'overlay ack o la fix finger call se intendiamo come risposta la fix finger response.
Prima di presentare il funzionamento di questo test a soglia è bene analizzare alcuni aspetti dello spazio delle chiavi. In [15] si evidenzia il fatto che la distanza fra un chiave casuale sull'anello e la chiave del successor di tale chiave è una geometrica di media pari alla distanza media fra nodi consecutivi. Questo è dovuto al fatto che la geometrica è una distribuzione senza memoria. Siccome le chiavi di destinazione dei nodi sono punti casuali sull'anello, allora la distanza response_distance fra la chiave k di destinazione di un messaggio e il relativo successor d è una variabile casuale identica alla distanza fra nodi. Formalmente response_distance è definita nel seguente modo: response_distance = d - k mod 2^m.
Vediamo ora come funziona la primitiva Distance Test. Quando un nodo onesto s manda un messaggio (applicativo oppure fix finger call), il nodo d su cui termina tale messaggio invia ad s una risposta. Il nodo s, ricevendo tale risposta, può allora calcolare response_distance = d – k mod 2^m ossia distanza tra la chiave del nodo d che ha inviato la risposta e la chiave di destinazione k del messaggio. Si osservi che il nodo s conosce k già da sé può scoprire il valore di d calcolando l'hash dell'indirizzo IP sorgente della risposta. Tanto response_distance è maggiore della distanza media fra due nodi allora l'overlay ack è più probabilmente falso perchè è più probabile che d non sia il vero successor della chiave k ma un nodo più avanti in senso orario nell'anello. Poiché le risposte false provengono dai maligni e non dagli onesti, se un overlay ack è falso allora proviene certamente da un nodo maligno. Se al contrario response_distance è paragonabile o inferiore alla distanza media fra due nodi, allora la risposta è più probabilmente generata dal vero successor della chiave di destinazione k il quale è onesto con probabilità 1 - f. In questo modo si cerca di rimuovere la componente Eclipse dell'attacco, facendo rimanere la ben più esigua componente Sybil. Dal

punto di vista algoritmico abbiamo tradotto i ragionamenti sopra espressi nel seguente modo: se response_distance > factor·distanza_media_nodi allora la risposta d è classificata falsa, mentre se response_distance ≤ factor·distanza_media_nodi allora la risposta è classificata vera. In altre parole, una realizzazione della distanza fra nodi è considerata tale se il suo valore è inferiore a factor volte il suo valore medio. Ciò è ragionevole poiché la distanza fra nodi è una geometrica, quindi i valori più probabili sono compresi fra zero e la media. Abbiamo aggiunto una costante reale "factor" poiché la distanza fra nodi, come prima detto, non è deterministica bensì una variabile casuale che solo in media è pari a: ring_circumference/N = 2^m/N. E' quindi possibile che via siano risposte provenienti da nodi onesti aventi response_distance superiore alla distanza media fra nodi. Questo fatto, ossia che la distanza fra nodi è non deterministica, dà luogo a falsi positivi FP e falsi negativi FN. Il tasso di FP e FN dipende naturalmente dalla soglia e per questo motivo la costante reale "factor" è un importante parametro che deve tarato opportunamente. Si può agire empiricamente in modo da minimizzare le percentuale di chiavi catturate oppure si possono raccogliere le distribuzioni di response_distance nei casi senza e con attacco e fissare factor in modo che factor·distanza_media_nodi in modo da minimizzare l falsi positivi o negativi.
La classificazione vera / falsa di una risposta serve a dedurre l'onestà / malignità del nodo d che risponde. Una risposta falsa proviene certamente da un nodo maligno in quanto, se il predecessor pointer di un nodo onesto è corretto, tale nodo onesto non fa terminare mai un messaggio di cui non è successor. Ciò implica che i nodi onesti non generano mai risposte relative a messaggi di cui non sono destinatari (responsabili). Al contrario un nodo maligno genera risposta falsa se il successor del messaggio diretto è onesto e questo messaggio è stato catturato da un maligno. In questo caso infatti il path del messaggio terminerà sul successor maligno del messaggio anziché sul vero successor che è onesto. Da questi motivi si conclude che una risposta falsa proviene solo da nodi maligni (nell'ipotesi che i predecessor pointers degli onesti siano corretti). Viceversa una risposta vera può venire sia da un maligno che da un onesto. Una risposta vera è generata da un onesto quando un messaggio non è catturato da nessun maligno e il vero successor del messaggio è un nodo onesto. In questo caso infatti il messaggio termina il proprio path sul nodo onesto che è successor della chiave di destinazione del messaggio. Una risposta vera proviene invece da un nodo maligno nel caso in cui il vero successor della chiave di destinazione del messaggio è un nodo maligno. In quest'ultimo caso la classificazione vera/falsa non aiuta affatto poiché la risposta è classificata vera e pertanto il nodo ritiene erroneamente che il messaggio diretto sia finito su un nodo onesto. Va notato tuttavia che un tale evento è raro nell'ipotesi che la percentuale di nodi maligni sia bassa. Se infatti f è la frazione di nodi maligni e nell'ipotesi che ogni nodo onesto generi G messaggi per unità di tempo e che questi siano indirizzati

uniformemente sull'anello, la frazione di messaggi il cui successor è maligno è NG(1-f)f / NG(1-f) = f cioè è pari alla frazione stessa dei nodi maligni (Sybil attack). Infatti, la probabilità che il successor di una chiave arbitraria sia un nodo maligno è infatti pari alla probabilità che un nodo qualsiasi nella rete sia maligno e questo è uguale alla frazione f di nodi maligni. L'altra situazione in cui la classificazione a soglia della risposta fallisce è quando response_distance è inferiore alla soglia factor·distanza_media_nodi. In questo caso il successor falso d della chiave k, non essendo sufficientemente lontano rispetto a k, appare secondo la soglia considerata come vero successor della chiave k (falso negativo).
E' interessante prestare attenzione al fatto che i maligni, per aggirare questa difesa, possono mettere una chiave sorgente falsa nella risposta. Infatti, come già citato, data la bassa percentuale di maligni è molto probabile che la response_distance sia superiore a factor·distanza_media_nodi, causando la classificazione falsa di tale risposta. Affinché al contrario la risposta falsa sia erroneamente classificata vera, per il maligno è opportuno inserire una chiave sorgente d tale che response_distance sia abbastanza piccola da essere minore di factor·distanza_media_nodi. Questa controdifesa dei maligni è possibile, in prima battuta, in caso di recursive routing poiché il nodo che riceve la risposta non sa l'indirizzo IP del maligno che ha originato la risposta dato che quest'ultimo ha percorso un overlay layer path e non un network layer path. Ricordiamo che è dall'indirizzo IP sorgente della risposta che si può ricavare la chiave d con cui si calcola response_distance. Nella nostra ipotesi del più efficiente semi-recursive routing, invece, ogni forma di risposta viene spedita direttamente al nodo interessato senza quindi essere soggetta ad alcun instradamento overlay. Il nodo maligno allora, per aggirare questo controllo, deve trovare un indirizzo IP tale che il suo hash dia una chiave d conveniente per lui ossia in modo che response_distance = $d - k \bmod 2^m$ sia comparabile o minore alla distanza media fra nodi. Dato che per ipotesi l'hash impiegato è crittograficamente sicuro, non essendo facilmente invertibile, il maligno è costretto calcolare l'hash di indirizzi IP random (forza bruta) finché non trova una chiave che rispetta la condizione di prima. A quel punto il maligno fa un IP spoofing sulla risposta mettendo l'indirizzo IP che dà la chiave desiderata. In questo modo l'hash da parte di chi riceve la risposta dà una chiave falsa così da ingannare il Distance Test. Possiamo suggerire due accorgimenti per impedire l'IP spoofing sulla risposta: 1) Impiegare TCP anziché UDP: impedisce l'IP spoofing sulla risposta in quanto il nodo che lo riceve vedrebbe che l'IP sorgente della risposta non appartiene alla connessione instaurata. 2) Se comunque si usa UDP il nodo che ha ricevuto la risposta può verificare l'esistenza del nodo sorgente della risposta attraverso un IP ping o un overlay ping (key ping). Un overlay ping è un applicazione di ping a livello overlay ossia usa messaggi a livello overlay. Se il nodo verificante s che manda l'echo request conosce già il socket del nodo di destinazione, glielo manda

direttamente (1 hop), altrimenti lo instrada sulla rete overlay come se fosse un qualsiasi messaggio applicativo che usa l'overlay. Nel nostro caso il nodo s invia l'echo request direttamente al nodo che deve sfidare poiché ne ricava il socket contenuto nella risposta (se l'implementazione di Chord in questione non prevede l'inserimento del socket come Chord header, il nodo verificante s può comunque ispezionare gli header IP e UDP). Siccome l'indirizzo IP contraffatto può essere scelto da un set random di indirizzi IP, esso molto verosimilmente non sarà l'indirizzo IP di un nodo della rete. Di conseguenza il ping non ottiene risposta entro un certo timeout e il nodo s classifica falsa la risposta in quanto contraffatta. Il maligno potrebbe allora aggirare questo controllo mettendo un indirizzo IP di un nodo n' onesto che succede abbastanza da vicino la chiave di destinazione k del messaggio (magari il suo vero successor!). Per evitare che il nodo s che ha ricevuto la risposta cada in questa trappola certamente non può usare l'IP ping (il nodo n' risponderebbe). Piuttosto, deve usare un overlay ping il cui messaggio echo request contenga un identificativo random univoco che era presente anche nel messaggio diretto chiave k che avrebbe dovuto ricevere, ossia quello che ha stimolato la risposta. In questo modo se il nodo s contatta il nodo n' con questo echo request il nodo n' vedendo che l'identificativo non coincide con l'identificativo di nessun messaggio ricevuto nell'ultima finestra temporale, potrebbe rispondere con un messaggio di errore opportuno. Perciò il nodo s si rende conto che il nodo n' il quale, secondo il punto di vista di s avrebbe mandato la risposta, in realtà non ha ricevuto il messaggio con chiave destinazione k che s aveva mandato. A questo punto il nodo s classifica la risposta come contraffatta e quindi l'esito del Distance Test sarà positivo, dunque la difesa reagirà di conseguenza. Questa soluzione che è necessaria con UDP è più complessa del caso TCP ma d'altronde il TCP non è certo la scelta migliore in termini di overhead per mandare un messaggio singolo quale è un messaggi di risposta. E' da precisare infine che, per evitare inconsistenze e soprattutto reply attack, la risposta deve contenere l'identificativo del messaggio che riscontra, così come l'overlay ping.

Infine precisiamo che il Distance Test non deve essere per forza associato ad una risposta, ma può tornare utile in tutti i casi in cui ci si aspetta che una certa grandezza sia una realizzazione della variabile distanza_media_nodi. Questo è il caso della difesa Delete Far Successors, nella quale la primitiva Distance Test è impiegata non a messaggi di risposta bensì sulle distanze fra entry consecutive della successor list.

In fig. 29 è riportato lo pseudocodice di Distance Test. E' da sottolineare che, siccome essa la usiamo come funzione per discriminare le realizzazioni della distanza media fra nodi da quelle delle distanze media fra nodi maligni, il suo impiego è più generale. Quello che vogliamo chiarire è che Distance Test può essere applicata anche ad una misura

(candidate_value) che non è response_distance, come ad esempio accade nella contromisura Delete Far Successors in cui l'input di Distance Test è la distanza fra entry consecutive della successor list e non response_distance.

```
if (candidate_value <= threshold)
    candidate_value is considered true
else
    candidate_value is considered false
```

Fig. 29: *Pseudocodice della primitiva di difesa Distance Test.*

Nodelist

L'idea è quella di avere in ogni nodo una lista di nodi random dell'anello (nodelist). Questa lista di nodi è poi usata nel lookup esattamente come la finger table e la successor list, ossia per trovare il closest preceding node della chiave di destinazione del messaggio. Il Chord lookup avviene secondo lo pseudocodice in fig. 30. Con il simbolo k intendiamo la chiave di destinazione del messaggio, mentre con successor(0) ci riferiamo alla prima entry della successor list, ossia al successor del node corrente che effettua il lookup.

```
if (k.isBetween(currentNode, successor(0)]) {
      next_hop = successor(0);
} else {
      next_hop = null;
      // successor's list lookup
      next_hop = successor_list.closest_preceding_node(k);

      // finger table's lookup
      if (finger_table.closest_preceding_node(k).isBetween(next_hop,k))
            next_hop = finger_table.closest_preceding_node(k);
}
```

Fig. 30: *Pseudocodice dell'algoritmo di lookup del protocollo Chord.*

Grazie alla nodelist si cerca di trovare un next hop che precede più da vicino la chiave di

destinazione k, aggiungendo in seguito al pseudocodice precedente il lookup della nodelist:

```
// nodelist's lookup
if (nodelist.closest_preceding_node(k).isBetween(next_hop,k))
      next_hop = nodelist.closest_preceding_node(k)
```

Fig. 31: *Pseudocodice dell'algoritmo di lookup della Nodelist.*

Passiamo ora al mantenimento della nodelist. Una policy ragionevole è aggiornarla come una coda FIFO: se l'inserimento di una nuova entry eccede la massima dimensione prefissata (max_queue_size), viene eliminata la entry più vecchia e aggiunta quella nuova. Infatti le entry più vecchie sono quelle meno utili in caso di churn poiché è più probabile che quelle vecchie siano uscite dalla rete che quelle più recenti. Il tempo medio di permanenza di un nodo nella nodelist è: max_queue_size/arrival_rate dove arrival_rate è la frequenza di inserimento dei nodi nella nodelist. La policy FIFO è la gestione più semplice ma si può anche pensare di eliminare per prima i nodi più prossimi al nodo n al fine di allungare il tempo di permanenza di quelli più lontani, i quali consentono di coprire meglio l'anello dove le finger non arrivano. La conoscenza dei nodi lontani è carente in Chord proprio per come è definita la finger table, infatti il nodo più lontano che un nodo n conosce è circa mezzo anello in avanti (è il successor della chiave $n + 2^{m-1}$), perciò la visione della rete di un nodo n non si spinge più lontano di metà anello. In conclusione questa difesa conferisce due vantaggi:

1) Permette di scavalcare i nodi maligni invalicabili. Per un nodo n un nodo maligno n' è invalicabile se il neighborhood di n e di tutti i nodi fra n e n' non contengono nodi onesti che sull'anello cadono dopo n'. In questo caso n' dà luogo ad un eclissi totale rispetto ai nodi compresi fra n (incluso) e n' perchè questi ultimi non hanno modo far arrivare messaggi ai nodi onesti che seguono n'. Una tale sfortunata situazione accade quando n' entra nella rete prima di tutti i nodi [n,n'). Ciò implica infatti che ogni nodo in [n,n') riceve dal proprio successor una successor list maligna poiché il primo nodo n_p in [n,n') che entra nella rete riceve la successor list da n', la quale è tutta maligna. Di conseguenza ogni nodo in [n, n_p) che entra riceverà una successor list maligna e anche ogni altro nodo che entra in (n_p,n'). Se dunque tutte le successor list dei nodi in [n,n') sono maligne, tutte le fix finger call emesse da tali saranno catturate dai maligni (n' o altri) e quindi anche le finger che puntano la parte di anello dopo n' saranno tutte maligne. Di conseguenza nessuna finger dei nodi [n,n') permette

di scavalcare n'. Se al contrario un nodo n_x in [n,n') grazie alla Nodelist, viene a conoscenza di uno o più nodi onesti che vengono dopo n', è possibile per tale nodo scavalcare n' mandando i messaggi con chiave (n', n_x) ai nodi onesti oltre n'. Si noti bene che la primitiva della Nodelist è l'unica soluzione per risolvere situazioni come queste, ovvero quando un nodo n non riesce a scavalcare un maligno n' nemmeno provando ad utilizzare esaustivamente tutti i nodi puntati dai nodi in [n,n'). Ma anche se ci fossero, come più di frequente accade, delle finger oneste che vengono dopo n' in qualche nodo in (n,n'), è comunque difficile che il messaggio alla fine superi n'. Sarebbe possibile con un inoltro ridondante da parte di ogni nodo ma ciò chiaramente darebbe origine ad un flooding, risultando poco efficiente. Da questi ragionamenti si evince da quanto sia in generale benefico disporre della Nodelist per andare al di là delle eclissi.

2) Riduce la lunghezza media del path. La nodelist, espandendo la conoscenza topologica della rete, permette al nodo di conoscere un maggior numero di nodi della rete anziché conoscere solo quelli contenuti nella finger table e nella successor list. Sapendo dell'esistenza di un maggior numero di nodi, si ottimizza in senso globale il lookup in quanto è possibile che nella Nodelist vi sia un nodo che precede più da vicino la chiave di destinazione k del messaggio rispetto al closest preceding node ottenuto dalla finger table e successor list. Di conseguenza i salti da nodo a nodi sono più lunghi e quindi saranno necessari meno nodi da attraversare prima di arrivare a destinazione. Il fatto di ottimizzare il routing riducendo l'hop count ha due benefici:
a) Diminuisce la probabilità che prima è poi il messaggio incontri un maligno, riducendo così la percentuale di messaggi catturati.
b) Decresce il ritardo end to end. Il fatto di ridurre l'hop count è buona cosa visto che in Chord il neighborhood in generale non è scelto in base alla caratteristiche prestazionali dei nodi e dell'underlay, quindi un nodo può puntare a nodi che hanno un ritardo elevato. Riducendo l'hop count può quindi portare a miglioramenti sensibili sul ritardo end to end.

Nelle pagine seguenti vi è la lista delle nostre soluzioni per difendere il protocollo Chord dall'attacco Eclipse.

6.3.1 Redundant First Hop

Funzionamento

In questa semplice soluzione il nodo sorgente s trasmette tante copie dello stesso messaggio su tutte le finger della finger table che precedono la chiave di destinazione. In questo modo aumenta la probabilità che vi sia un path che termini sul successor onesto del messaggio, se esiste. Per messaggio qui intendiamo sia i messaggi applicativi che le fix finger call. Redundant First Hop è una soluzione dispendiosa in termini di traffico ma comunque meno del caso in cui anche i nodi intermedi adottassero la stessa trasmissione multipla. Abbiamo sperimentato questa difesa per esplorare la possibilità di difendersi dall'attacco Eclipse semplicemente sfruttando le sole informazioni di routing previste da Chord.

Overhead di traffico

Ogni messaggio anziché essere trasmesso una volta, viene trasmesso n_{tx} volte dal nodo sorgente. Se la chiave di destinazione k del messaggio cade dopo l'ultima finger, allora la trasmissione interessa tutta la finger table e perciò si ha $n_{tx} = O(\log_2 N)$ infatti $O(\log_2 N)$ è il numero di finger. Se invece k sta fra la prima e la seconda finger allora il messaggio è inoltrato solo alla prima finger e quindi $n_{tx} = 1$.

Overhead di memoria

Nessuno.

Overhead computazionale

Nessuno.

6.3.2 Alternative First Hop

Funzionamento

E' una versione più efficiente della Redundant First Hop dove infatti ad ogni messaggio, il nodo sorgente analizza la relativa risposta e reagisce in base ad essa. L'idea è infatti quella di reagire con la scelta di un path diverso per ogni messaggio che sembra essere terminato su un nodo maligno. Con messaggio intendiamo sia i messaggi applicativi che le fix finger

call. La risposta di una fix finger call è la fix finger response mentre la risposta di un messaggio dati è un overlay ack. Questa difesa funziona nel seguente modo: quando la risposta di un messaggio è classificata falsa (tramite la primitiva Distance Test), oppure non arriva entro un certo timeout, tale messaggio viene ritrasmesso sulla finger i – current_attempt dove i è la finger usata nel primo tentativo, ossia la finger secondo il normale Chord routing. L'indice current_attempt va da 1 min(max_attempt, finger_table_size), mentre il parametro max_attempt è libero ed esprime il massimo numero di finger impiegate per le ritrasmissioni. Si comprende quindi che i tentativi di ritrasmissione di un messaggio consistono nel trasmettere lo stesso messaggio sulle finger via via precedenti la finger che verrebbe scelta secondo il Chord lookup. Non avrebbe senso infatti usare le finger successive perché ciò darebbe luogo ad un giro completo dell'anello, il quale quasi certamente implica che il messaggio sia carpito dal subring maligno.

Overhead di traffico

Si hanno tre contributi:

1) Per ogni messaggio applicativo è esplicitamente necessario il relativo overlay ack, il quale tuttavia è privo di overlay payload in quanto messaggio di controllo di overlay.

2) Ritrasmissione dei messaggi applicativi la cui risposta è classificata falsa (o non arriva entro un timeout). Se d_m è la dimensione di un messaggio applicativo, allora una singola trasmissione senza ack costa $C_t = d_m$. Con l'Alternative First Hop il costo per trasmettere un messaggio applicativo diventa: $C_t = E[n_t](d_m+d_a)$ dove d_a è la dimensione dell'ack e $E[n_t]$ è il numero medio di trasmissioni del messaggio. L'overhead di traffico quindi si attesta su un fattore $E[n_t](d_m+d_a)/d_m$. Per quanto concerne $E[n_t]$ si ha: $E[nt] = \sum_1^{\max\_attempt} kp(k)$ con k da 1 a max_attempt e p(k)=probabilità che venga effettuata la trasmissione k = probabilità che le trasmissioni da 1 a k - 1 abbiano dato timeout o risposta classificata falsa. Notare che p(k) dipende dal path effettuato dal messaggio del tentativo k e ciò dipende dalla posizione del nodo sorgente, dei nodi che stanno tra quest'ultimo e il nodo di destinazione e dal loro inquinamento.

3) Ritrasmissione delle fix finger call. Senza questa difesa, per costruire una finger, si invia una fix finger call e si riceve la relativa fix finger response. Pertanto il costo per finger è $C_t = d_{fc} + d_{fr}$ dove d_{fc} = dimensione fix finger call, mentre d_{fr} = dimensione fix finger response. Con questa difesa invece il costo per finger è $C_t = E[n_t](d_{fc} + d_{fr})$ e vale la stessa espressione di prima per $E[n_t]$. Si osservi infine che le fix finger call / response sono pacchetti molto più piccoli dei messaggi applicativi poiché, essendo messaggi di controllo, non portano nessun overlay payload.

Overhead di memoria

Non ci sono strutture dati aggiuntive. L'unico memoria addizionale riguarda il buffering per le ritrasmissioni.

Overhead computazionale

Abbiamo due componenti:

1) Gestione delle ritrasmissioni

2) Classificazione delle risposte

Entrambe danno un overhead trascurabile.

6.3.3 Delete Far Successors

Funzionamento

Nella descrizione di Chord avevamo visto che un nodo n inizializza la propria successor list a seguito della ricezione della successor list del bootstrap node che ha contattato nel momento in cui il nodo n è entrato nell'overlay Chord. Tale successor list viene ricevuta dal nodo n tramite un messaggio di join response. Avevamo anche visto che il nodo n aggiorna periodicamente la sua successor list ricevendo quella del suo successor mediante un messaggio di notify response.

L'idea della difesa Delete Far Successors è quella di eliminare, da parte del nodo n, le entry classificate false della successor list ricevuta con la notify response. La successor list è infatti per definizione una lista di nodi consecutivi e per questa ragione ci si aspetta che la distanza fra due entry consecutive siano valori di una geometrica il cui valor medio è la distanza media fra nodi, ossia $2^m/N$. Questo è vero se non c'è attacco Eclipse. Se c'è attacco Eclipse le cose cambiano: la successor list di un nodo maligno contiene infatti i soli nodi maligni consecutivi. Poiché assumiamo verosimilmente che i nodi maligni siano uniformemente distribuiti sull'anello, allora non capita mai che tutte le entry siano nodi consecutivi. Se ad esempio nella rete vi è una frazione di f nodi maligni, allora la distanza media fra di essi è $2^m/fN$, la quale è di $1/f > 1$ volte maggiore della distanza media fra nodi generici. Per questo motivo la distanza media fra le entry della successor list di un nodo maligno è una geometrica la cui media è $1/f > 1$ volte maggiore della media della distanza fra le entry di una successor list che si avrebbe senza attacco. Poiché i nodi maligni, esattamente come gli onesti, danno agli altri nodi la propria successor list (join response / notify response), anche nella successor list degli onesti figurano sequenze di nodi maligni, ovvero sequenze di entry

la cui distanza è mediamente 1/f volte superiore alla distanza fra due entry oneste. La difesa Delete Far Successors funziona pertanto nel seguente modo: un nodo n quando riceve una successor list, calcola la distanza fra tutte le coppie di entry consecutive, perciò ottenendo succlist_size – 1 distanze. Se la distanza entry(i) – entry(i – 1) (con 1 ≤ i ≤ succlist_size) è superiore a factor·distanza_media_nodi allora la entry(i) è ritenuta falsa cioè non è ritenuta successor della entry precedente entry(i - 1). Questo porta a supporre che il nodo entry(i) sia maligno e dunque il nodo n non inserisce tale entry(i) nella sua successor list. In conclusione questa difesa ha l'obbiettivo di prevenire l'inquinamento della successor list eliminando dalla successor list ricevuta le entry ritenute maligne. Si noti infine che non è possibile applicare questo algoritmo sulla successor list ricevuta al momento della join poiché, non conoscendo ancora la distanza media fra nodi, non è possibile invocare la primitiva Distance Test. In fig. 32 è mostrato lo pseudocodice di questa contromisura. Il simbolo successor(0) indica la prima entry della successor list, ossia il successor del nodo corrente.

```
interval = successor(0) – current_node     mod 2^m
if (DistanceTest(interval).isTrue)
      new_successorlist.insert(successor(0));
for (k = 1; k < succlist_size; k++) {
      interval(k) = successor(k) – successor(k-1)    mod 2^m
      if (DistanceTest(interval(k)).isTrue)
            new_successorlist.insert(successor(k));
}
successorlist = new_successorlist;
```

Fig. 32: *Pseudocodice di Delete Far Successors.*

Overhead di traffico

Nessuno perchè non implica la trasmissione di messaggi.

Overhead di memoria

E' trascurabile in quanto non si ha nessuna struttura dati addizionale.

Overhead computazionale

Riguarda semplicemente una semplice analisi della successor list ogni volta che deve essere aggiornata e ciò avviene ad ogni notify response, quindi a seguito di una stabilize la quale avviene ad ogni stabilize_interval. Quest'ultimo è un parametro modificabile e

nell'implementazione di OverSim è per default 20 s.

6.3.4 Augmented Fix Finger

Funzionamento

Consideriamo un nodo n che deve costruire o fare il refresh periodico della finger table. Per far ciò, in base alle regole di Chord, questo nodo n invia m messaggi di fix finger call e inserisce come finger corrispondenti i nodi sorgenti delle relative fix finger response.
Con la difesa Augmented Fix Finger, al metodo tradizionale di Chord, si affianca un sistema di messaggi, secondo lo stesso modello call / response, avente lo scopo di ottenere altri nodi candidati oltre a quelli ottenuti con le fix finger response. Prima di tutto ricordiamo che per neighborhood di un nodo j intendiamo tutti i nodi direttamente raggiungibili dal nodo j (finger table, successor list e più eventualmente la nostra Nodelist). Secondo questa nuova tecnica, un nodo n costruisce la generica finger i ($0 \leq i \leq m - 1$) in tre passi, il primo dei quali già presente in Chord, gli altri due sono i nuovi:
1) Fix Finger Call. Il nodo n manda un messaggio di fix finger call avente chiave $n + 2^i$, esattamente come da protocollo Chord.
2) Neighborhood Collection. Quando la rispettiva fix finger response arriva al nodo n, quest'ultimo anziché inserire il nodo sorgente di essa come finger i, manda un messaggio di neighborhood call all'ultima finger j ottenuta prima della finger i. Il nodo corrispondente alla finger j risponde allora direttamente al nodo n con un messaggio di neighborhood response contenente i socket del suo neighborhood. Il nodo n quindi apprende la conoscenza di questi nuovi nodi e li memorizza temporaneamente assieme a quelli ottenuti in precedenza interpellando con la neighbor call le finger ottenute prima della finger j. Chiamiamo con S(j) questo gruppo di nodi appreso dalle neighbor response mandate dalla finger j e da quelle precedenti.
3) Finger Choice & Set. A questo punto il nodo n, allo scopo di inserire la nuova finger i, anziché avere a disposizione un solo nodo candidato come avverrebbe normalmente in Chord (che sarebbe la fix finger response), ha a disposizione di una moltitudine di candidati. Questi candidati sono i nodi del gruppo S(j) in aggiunta naturalmente al nodo sorgente della fix finger response. Di conseguenza, il nodo n avendo a disposizione più candidati anziché un solo, ha il vantaggio di poter scegliere quello che rispetta al meglio le regole del protocollo. Ciò si traduce nel fatto che il nodo n sceglie come finger i il nodo che segue più da vicino la chiave $n + 2^i$, poiché la finger i per definizione è il successor della detta chiave.

Questo sistema ampliato di ottenimento delle finger ha il beneficio di ridurre la probabilità che nella finger table entri una finger falsa, ovvero finger che non è il successor della chiave della fix finger call a cui aveva risposto. Siccome le finger false sono finger maligne, si comprende come questo metodo riduca la percentuale di finger maligne.
I tre step sopra descritti, ossia l'algoritmo Augmented Fix Finger, viene ripetuto per ognuna delle m fix finger call / response finché la finger table non è terminata. L'aggiornamento della finger table si ritiene terminato quando il nodo riceve l'ultima fix finger response, ossia quella con offset 2^{m-1} rispetto alla chiave del nodo. Inoltre si noti che la lista S(j) dei socket dei nodi guadagnati con lo step 2 cresce da iterazione a iterazione finché non viene svuotata non appena la finger table è stata completata.
E' interessante osservare che tra tutte le difese qua proposte questa è l'unica che comporta uno scambio apposito di messaggi fra nodi. Qui infatti si tratta di andare a definire un protocollo che consente ad una nodo n di chiedere la finger table e la successor list (neighborhood) di un altro nodo q. E' tuttavia vero che il nodo n può ricostruire autonomamente la finger table di un altro nodo q, infatti il nodo n può calcolarsi da sè tutti gli start del nodo q (sono $q + 2^i$, $0 \leq i \leq m - 1$) però non può ricostruire la sua successor list perchè essa può essere ricostruita solo dalla join response e notify response ricevute dal solo nodo q.

Overhead di traffico

Senza difesa la costruzione della finger table costa $m \cdot (d_{fc} + d_{fr})$ [byte] dove d_{fc} e d_{fr} sono rispettivamente la dimensione della fix finger call e response. Con la difesa, a questo traffico dovuto alle fix finger call / response, si aggiunge quello dovuto alle neighborhood call / response. Il nodo infatti invia un certo numero di messaggi neighborhood call così da ricevere altrettanti messaggi di neighborhood response. Visto che per ogni neighbor vi è una coppia di neighborhood call / response, e che i neighbor interpellati sono ft_length – 1, anche le coppie di neighborhood call / response sono ft_length – 1. Per ft_length intendiamo il numero di finger che la finger table avrà alla fine della sua costruzione. Non sono interpellate tutte le finger, infatti l'ultima non lo è perché le sue informazioni di neighborhood non sono utili poiché la costruzione della finger table è terminata. Pertanto si aggiunge il costo $C_t = (ft\_length - 1) \cdot (d_{nc} + d_{nr})$ essendo d_{nc} e d_{nr} la dimensione di un messaggio di neighborhood call e response. Di conseguenza il traffico incrementa di un fattore $((ft\_length - 1) \cdot (d_{nc} + d_{nr}) + m \cdot (d_{fc} + \cdot d_{fr})) / m \cdot (d_{fc} + \cdot d_{fr})$ essendo $m \cdot (d_{fc} + \cdot d_{fr})$ il costo in byte della costruzione della finger table secondo il metodo di Chord.
Analizziamo ora le dimensioni dei messaggi. Un messaggio di neighborhood call non ha payload quindi la sua dimensione è assimilabile a quella di una fix finger call o response.

Non essendoci overlay payload, questi tre messaggi sono trasportati da un solo pacchetto IP. Un neighborhood response, invece, contiene i socket dei nodi presenti nella successor list e nella finger table del nodo che risponde, pertanto il payload è costituito da (succlist_size + ft_length)·socket_size. Ricordiamo che il numero di entry della successor list (succlist_size) è un parametro libero (lo abbiamo fissato a 16 entry) mentre il numero di finger (ft_length) è proporzionale al numero di nodi nella rete ed è al max pari a m (lunghezza chiave in bit). In genere il payload di una neighborhood response è abbastanza basso da non richiedere la frammentazione in più pacchetti IP. Nel casi peggiori, cioè nel caso di un numero di nodi N molto elevato, ad esempio $N = 10^7$ allora la finger table ha circa $\log_2(10^7) = 23$ entry. Se supponiamo succlist_size = 16, allora il payload è $(23 + 16)\cdot 6 = 312$ byte che non implica la frammentazione né di UDP né di IPv4/v6. Come avevamo detto nel capitolo di Chord, ipotizziamo che tutti i messaggi di controllo Chord sono trasportati da UDP visto che si tratta di pochi messaggi isolati nel tempo e indirizzati verso tanti nodi differenti e quindi non sarebbe efficiente usare TCP a causa dell'instaurazione della connessione.

Overhead di memoria

Gestione dell'invio e della ricezione dei messaggi di neighborhood call e response.

Overhead computazionale

Gestione dell'invio e della ricezione dei messaggi di neighborhood call e response.

6.3.5 Neighbor Challenge

Funzionamento

In [2] allo scopo di rilevare partizionamenti nell'anello, viene suggerita la proposta in cui un generico nodo n, invia periodicamente query aventi chiave di destinazione il nodo n stesso. Se la rete Chord è corretta, il nodo n si aspetta di riceve tale query poiché esso stesso è il responsabile della chiave n. Se invece vi sono nodi il cui puntatore al successore è sbagliato oppure nodi anomali che eliminano i messaggi, allora si possono verificare situazioni di errore in cui tale query (o un messaggio in genere) entra in un loop senza giungere mai al rispettivo nodo successor (responsabile). Oppure, come detto prima, la query non giunge mai a destinazione perchè catturata dal subset dei nodi collusi. Nel caso dell'attacco Eclipse una tale query, se viene catturata da un maligno, allora non può tornare al nodo stesso che l'ha mandata, infatti termina sul successor maligno della query. Per il nodo n sarebbe

allettante sfidare i nodi da lui conosciuti (neighborhood) inviando periodicamente una query di chiave n. Questo però non conduce a nessun risultato plausibile. Se infatti il nodo n inoltra una query di chiave n ad un nodo da lui conosciuto n' (una finger o una successor list entry), se la query arriva al nodo n allora significa che tutti i nodi attraversati dalla query hanno instradato correttamente (e perciò anche il nodo n'), in caso contrario significa che un nodo lungo il path non ha instradato correttamente. In caso di iterative routing il nodo sorgente n è chiaramente in grado di identificare il nodo che elimina la query (nel caso di dropping) e potrebbe anche dedurre quel è il nodo che non instrada correttamente (nel caso di bad routing). Nel caso invece di semi recursive routing (quello che assumiamo se non specificato) o di recursive routing, il nodo sfidante non può sapere se l'errore è stato causato dal suo neighbor n' o da un altro. Se il nodo n' non è l'ultimo hop nel path corretto e la query non arriva al nodo n allora il comportamento anomalo potrebbe essere dovuto sia al nodo sfidato n' che ad un altro nodo tra n' e n. Siccome il nodo n in generale non sa a priori il path corretto che dovrebbe seguire la query, non può nemmeno sapere se il nodo n' è l'ultimo hop del path corretto della query. Di conseguenza il mancato arrivo della query può essere imputato, oltre ad n', ad un qualunque nodo tra n' e n. Per questo motivo il mancato arrivo della query, in generale, non conduce alla conclusione che il nodo sfidato n' sia non corretto. Sempre mantenendo il concetto di sfidare i nodi conosciuti dal generico nodo n, la nostra proposta di difesa è piuttosto la seguente: il nodo n sfida un nodo n' (un suo neighbor) inviando una query la cui chiave è di responsabilità del successor del nodo n'. In questo caso il nodo n' inoltra direttamente tale query al suo successor il quale ritorna la risposta al nodo sorgente n della query. Il nodo n può allora calcolare response_distance e classificare la risposta come vera / falsa tramite la funzione Distance Test. Se la risposta è vera significa che il successor del nodo n' è il vero successor di n', altrimenti no. Il nodo d non è il vero successor di n' se n' è maligno e il suo successor maligno non coincide con il suo vero successor (altamente probabile se i maligni sono pochi e sparsi). Se dunque il nodo classifica la risposta falsa, allora ha trovato un modo per classificare il nodo n' come maligno. Questo principio può essere sfruttato dal nodo n per sfidare, a sua discrezione, ogni suo puntatore ad altri nodi (finger o successor list entry). Se il nodo sfidato è classificato maligno, il nodo quindi lo elimina dal suo neighborhood riducendone così il grado di inquinamento.

Overhead di traffico

Per ogni neighbor sfidato vi è un messaggio di challenge query e la relativa risposta (ad esempio l'overlay ack). La challenge query e la risposta non hanno overlay payload, quindi sono un pacchetti di piccole dimensioni.

Overhead di memoria

Nessuno.

Overhead computazionale

Nessuno.

6.3.6 Blacklist

Funzionamento

Come abbiamo visto, le risposte possono essere soggette ad una classificazione binaria (vera/falsa) tramite la primitiva Distance Test. Quando al nodo n arriva una risposta e la classifica falsa, il nodo n inserisce il nodo sorgente d di tale risposta in una lista di nodi (blacklist). Così facendo, ogni volta che viene aggiornata la successor list e la finger table, si esclude la possibilità di inserire tale nodo d sia nella successor list che nella finger table. Ovviamente più la blacklist è piena di nodi maligni e meno saranno i nodi maligni che riusciranno ad infilarsi nella finger table e nella successor list. Si potrebbe dunque proporre una versione cooperativa di questa difesa anziché esclusivamente locale: i nodi potrebbero scambiarsi messaggi appositi in cui pubblicano i nodi classificati maligni così da massimizzare il riempimento delle proprie blacklist. E' ovvio che quest'ultima soluzione presenta la vulnerabilità per la quale se un nodo maligno n' manda un messaggio contenente nodi onesti ad un nodo onesto n che non ha ancora classificato n' come maligno, il nodo n inserisce nella propria blacklist i nodi onesti pubblicati da n'.
In fig. 33 è mostrato l'algoritmo di inserimento di un nodo nella blacklist.

```
response_distance = d – k mod 2^m
if (DistanceTest(response_distance).isFalse)
        Blacklist.insert(IP_address_d);
```

Fig. 33: *Pseudocodice dell'inserimento di un nodo nella blacklist.*

Overhead di traffico

E' richiesto un overlay ack per ogni messaggio terminato.

Overhead di memoria

La blacklist è di per sé una struttura dati aggiuntiva. Supponiamo che una entry di blacklist sia l'indirizzo IP del relativo nodo classificato maligno. Se la blacklist è implementata attraverso allocazione dinamica, allora la sua occupazione varia da 0 (nessun nodo classificato maligno) ad un massimo di IP_address_size·max_queue_size dove max_queue_size è il numero max di entry. Si osservi che a differenza della Nodelist, qui basta inserire il solo indirizzo IP, infatti è sufficiente un identificativo univoco dei nodi vietati. Nel caso della Nodelist invece serve anche il numero di porta perché essa contiene nodi che devono essere direttamente contattati. Anche qui vale lo stesso discorso fatto per la Nodelist e cioè, quando si deve inserire un nuovo nodo nella blacklist ed essa ha già raggiunto la dimensione massima, è ragionevole rimuovere quella più vecchia (FIFO queue). Siccome ci si aspetta che il numero di nodi maligni sia verosimilmente piccolo, è ragionevole fissare max_queue_size ad un valore di qualche decina o centinaio di nodi e quindi si avrà un'occupazione di appena qualche decina di KB. Ad esempio se vi sono 200 nodi nella blacklist, la memoria richiesta è 6·200 = 1200 bytes = 1.17 KB, avendo assunto 6 byte di socket size.

Overhead computazionale

E' costituito dai seguenti contributi:

1) Ricerca di elementi comuni fra la successor list e la blacklist. Tale operazione avviene ad ogni aggiornamento della successor list che avviene in media ogni stabilize_interval, il quale è un parametro arbitrario (in OverSim è di default 20s).

2) Controllo che il nodo sorgente di una fix finger response non sia presente nella blacklist. La frequenza di questa operazione è pari alla frequenza con cui sono ricevute le fix finger response. Esse sono pari a m ogni fix_finger_interval quindi tale frequenza è m/fix_finger_interval dove fix_finger_interval è un parametro regolabile e nell'implementazione di OverSim è pari a 100s di default.

6.3.7 Whitelist

Funzionamento

Se mediante la primitiva Distance Test una risposta risulta classificata vera, allora è molto più probabile che essa giunga da un onesto che da un maligno. E' allora conveniente memorizzare questi nodi in una Nodelist (whitelist perchè sono nodi classificati onesti).

Questa sorta di cache è quindi una declinazione della primitiva Nodelist. Si veda tale paragrafo per quanto riguarda la sua modalità d'uso, il suo mantenimento e i vantaggi del suo impiego. In fig. 34 è mostrato lo pseudocodice della whitelist.

```
response_distance = d – k mod 2^m
if (DistanceTest(response_distance).isTrue)
        Nodelist.insert(socket_d);
```

Fig. 34: *Pseudocodice di Whitelist.*

Overhead di traffico

Per ogni messaggio applicativo giunto a destinazione si richiede un overlay ack.

Overhead di memoria

Come la blacklist anche la Nodelist è di per sé una struttura dati aggiuntiva. Non c'è ragione per cui l'implementazione della nodelist dev'essere differente da quella della blacklist per cui valgono le stesse considerazione fatte per quest'ultima. L'unica differenza è che, essendo più probabile che nella rete vi siano più nodi onesti che maligni, è preferibile fissare il limite di dimensione della Nodelist ad un valore più alto della blacklist.

Overhead computazionale

Ad ogni lookup viene consultata anche la nodelist per trovare il closest preceding node della chiave del messaggio da inoltrare.

6.3.8 Source Enhanced Local Info

Funzionamento

L'idea consiste da parte di un nodo onesto n di leggere gli indirizzi IP sorgenti contenuti nei messaggi che riceve. Essi sono i seguenti:

1) L’indirizzo IP sorgente del pacchetto che incapsula il messaggio Chord ricevuto: poiché questa è un’informazione “gratuita”, ossia già presente nei pacchetti che riceve, per il software Chord è sufficiente accedere all’header IP del pacchetto per guadagnare questa informazione di routing.

2) L’indirizzo IP dell’originatore del messaggio Chord ricevuto: siccome ad ogni overlay hop l’indirizzo IP sorgente del pacchetto cambia (quello del punto 1), è necessario esplicitamente inserire tale indirizzo IP nel Chord header. Questo non è troppo costoso in termini di banda ed è comunque necessario per permettere la risposta diretta del messaggio (overlay ack, fix finger response).

Lo scopo di leggere questi due indirizzi IP da parte di un nodo è quello apprendere, senza richieste esplicite, l'esistenza di nodi che non fanno già parte del suo neighborhood. Dall'hash dei due indirizzi IP menzionati si ottengono poi le chiavi dei rispettivi nodi. Quando un nodo riceve un messaggio, per ognuno dei due indirizzi IP, detta s la chiave derivata dall’indirizzo IP, è eseguita una delle due operazioni seguenti:

1) Ottimizzazione della finger table. Il nodo n scansiona la finger table in cerca di una entry $0 \leq i \leq m - 1$ per la quale la chiave s cade nell'intervallo [start(i) e finger(i)). Se una tale entry esiste, viene sostituita la finger(i) con il nodo rappresentato dalla chiave s. Dal momento che la maggior parte delle entry maligne sono caratterizzate da un ampio intervallo [start(i) e finger(i)), è molto probabile che il nodo di chiave s cada in quell'intervallo rimpiazzando così la finger maligna finger(i).

2) Se il nodo s non viene inserito né nella finger table, cioè nel caso in cui s non serva ad ottimizzare la finger table, il nodo n memorizza comunque s in una Nodelist che viene utilizzata assieme alla finger table e alla successor list ad ogni lookup. Questo accorgimento permette di espandere l'informazione di routing da parte del nodo n, aumentando la probabilità, ad ogni lookup, di inoltrare il messaggio verso un nodo più prossimo alla destinazione di tale messaggio.

Discutiamo ora le possibili vulnerabilità di questa difesa. Nell’attacco Eclipse i nodi maligni non trasmettono mai messaggi verso gli onesti, per cui i nodi sorgenti osservati sono sempre onesti. Tuttavia i nodi maligni possono comunque disturbare questa difesa inviando tanti messaggi a random sui nodi da esso conosciuti in modo che le due chiavi ottenute (maligne) da quei messaggi siano viste da più nodi possibili. Tuttavia se la rete è grande e i nodi maligni sono relativamente pochi, anche il numero di nodi onesti da essi conosciuti sono pochi, e quindi l'impatto di questa azione sarebbe limitata. Anche nell'ipotesi in cui l'attaccante conoscesse tutti i nodi onesti nella rete, sarebbe oltremodo dispendioso in termini di risorse, specialmente di banda. Ancora, i nodi maligni, possono creare rumore sull'esistenza dei nodi generando tanti indirizzi IP casuali (da cui si hanno le chiavi) e facendo IP spoofing su questi messaggi di rumore. Il nodo onesto n per difendersi da questo rumore, prima di impiegare una chiave sorgente per l'ottimizzazione, deve verificare che essa corrisponda ad un nodo esistente (es. ping IP oppure un overlay ping).

La fig. 35 illustra lo pseudocodice di questa difesa, la quale è una funzione chiamata, in

teoria, ad ogni messaggio ricevuto.

```
for (i = 0; i < ft_length; i++) {
      if (key_1.isBetweenL(start(i), finger(i).key))
         finger(i) = socket_1;
      else
         Nodelist.insert(socket_1);
      if (key_2.isBetweenL(start(i), finger(i).key))
         finger(i) = socket_2;
      else
         Nodelist.insert(socket_2);
}
```

Fig. 35: *Pseudocodice di Source Enhanced Local Info.*

Overhead di traffico

Nessuno.

Overhead di memoria

I nodi sorgenti s di ogni messaggio che non sono inseriti nella routing table e nella successor list, sono memorizzate in una Nodelist. Ogni entry è un socket.

Overhead computazionale

Si hanno due contributi:

1) Ricerca nella finger table di una entry nel cui intervallo [start,finger) cade la chiave del nodo sorgente del messaggio. La scansione della finger table costa ft_length = $O(\log_2 N)$ confronti numerici e avviene ad ogni messaggio ricevuto.

2) Se la nodelist non è vuota, ad ogni lookup, viene consultata esattamente come la finger table e successor list. Il lookup della Nodelist costa O(nodelist_size) confronti numerici e se ipotizziamo che ogni messaggio ricevuto abbia destinazione uniforme sull'anello e che la nodelist sia un campionamento uniforme e ordinata per chiave, allora in media si hanno nodelist_size/2 confronti.

6.3.9 Path Enhanced Local Info

Funzionamento

E' come la proposta precedente, con l'unica differenza che ora in un messaggio si appende tutto il path percorso dal messaggio. In altre parole, ogni nodo che riceve un messaggio, inserisce il proprio socket nell'overlay header e quindi lo reinstrada. La conseguenza è che ogni nodo, osservando i messaggi ricevuti da esso, anziché apprendere l'esistenza del solo nodo sorgente, vede anche il path finora effettuato e quindi i nodi che sono stati attraversati dal messaggio. Per il resto questa ottimizzazione è identica alla Source Enhanced Local Info. Anche qui valgono le stesse considerazioni circa le possibili vulnerabilità. La fig. 36 illustra lo pseudocodice di questa difesa, la quale è una funzione chiamata, in teoria, ad ogni messaggio ricevuto.

```
for (i = 0; i < ft_length; i++) {
      for (k = 0; k < hop_count; k++) {
         if (key(k).isBetweenL(start(i), finger(i).key))
               finger(i) = socket(k);
         else
               Nodelist.insert(socket(k));
       }
}
```

Fig. 36: *Pseudocodice di Path Enhanced Local Info.*

Overhead di traffico

Ogni nodo che inoltra un messaggio appende il suo socket. La lunghezza media di un path in Chord è $E[h] = (1/2)\cdot\log_2(N)$, quindi per una rete di N = 1000 nodi si ha E[h] = 5 hop, che risulta in un overhead (socket IPv4) di 5·6 = 30 byte, mentre per $N = 10^6$ nodi si ha circa E[h] = 10 hop che risulta in 10·6 = 60 byte.

Overhead di memoria

I nodi del path di ogni messaggio che non sono inseriti nella routing table e nella successor list, sono memorizzate in una Nodelist. Qui l'hop count parziale medio è ovviamente inferiore all'hop count totale $(1/2)\log_2(N)$. Al solito, la Nodelist può essere implementata con una struttura di memoria dinamica in cui si definisce un limite massimo di dimensione.

Overhead computazionale

Si hanno due contributi: 1) Per ogni hop del path, si esegue la ricerca nella finger table di una entry nel cui intervallo [start,finger) cade la chiave del nodo sorgente del messaggio. 2) Se la nodelist non è vuota, ad ogni lookup, viene consultata esattamente come la finger table e successor list. Qui valgono gli stessi discorsi fatti per la proposta Source Enhanced Local Info.

6.3.10 Anti Shield Effect

Funzionamento

In Chord se vi è un messaggio con chiave di destinazione pari a quella di un nodo esistente k e tale nodo k è presente nella finger table, il nodo non inoltra tale messaggio direttamente a k, bensì al suo closest_preceding_node.

Con questa proposta, invece, il nodo spedisce direttamente il messaggio al successor di tale messaggio, ossia il nodo k. Dal punto di vista algoritmico ciò si traduce nel fatto che la if(finger(i) (n,id)) dell'algoritmo di Chord [2] diventi if(finger(i) (n,id]). E' da sottolineare inoltre che anche il lookup della successor list avviene secondo tale algoritmo. Questa ottimizzazione di routing si rivela preziosa per aggirare lo "shield effect" di Chord, ossia al fatto che un messaggio, prima di terminare sul suo successor, passa necessariamente per il suo predecessor. Se l'attaccante avesse sufficiente indirizzi IP da cui scegliere, esso potrebbe scegliere quello il cui hash (la chiave del nodo) gli consente di ottenere un nodo che sia il predecessor di un nodo vittima n. In questo modo l'attaccante sarebbe in grado di intercettare ogni messaggio di cui il nodo vittima n è responsabile. Con questa modifica invece, i messaggi di cui il nodo n è responsabile non devono più passare necessariamente per il suo predecessor.

Questa ottimizzazione non aggiunge assolutamente alcun onere computazionale né di memoria e ha l'effetto di ridurre la lunghezza del path quando una chiave giunge ad un nodo e tale nodo ha una finger o una successor list entry pari a tale chiave. Siccome questa ottimizzazione non costa nulla, l'abbiamo tenuta sempre presente in tutte le simulazioni in cui sono valutate le singole difese o i set di difesa.

Overhead di traffico

Nessuno.

Overhead di memoria

Nessuno.

Overhead computazionale

Nessuno.

6.3.11 Counterclockwise Routing

Funzionamento

L’idea è quella di aumentare la complessità spaziale avendo due finger table e due successor list anziché una sola, allo scopo di ridurre la lunghezza dei path più lunghi in cui la chiave di destinazione cade nell’altra metà dell'anello. L'idea è quella di sfruttare al meglio la doppia informazione topologica che ora ha disposizione il nodo: si seleziona il neighbor che, tra quelli orari e antiorari, è più prossimo alla chiave di destinazione. Ciò riduce l'hop count medio del path e quindi la probabilità di cattura. Oppure si può mandare lo stesso messaggio in due copie, una in senso orario e l’altra in senso antiorario. Notare che il principio di incrementare la complessità spaziale per ridurre quella temporale (lunghezza path) è lo stesso principio della Extended Finger Table, della Whitelist e della External Nodelist. Per motivi di tempo non abbiamo implementato questa soluzione, in verità abbastanza radicale.

Overhead di traffico

L’overhead in termini di traffico raddoppia per ciò che riguarda le fix_finger call / response poiché vi sono due finger table da mantenere. Dato che qui non c’è nessun overlay ack per i messaggi applicativi, non si ha nessun overhead sul traffico applicativo. Esso raddoppierebbe solo nel caso in cui si duplica la trasmissione del messaggio, una in senso orario e l'altra in senso antiorario.

Overhead di memoria

Siccome vi sono due finger table, la complessità di memoria semplicemente raddoppia (passa da $O(\log_2 N)$ a $O(2\log_2 N)$) e pertanto rimane logaritmica.

Overhead computazionale

Abbiamo due componenti:

1) Raddoppio della computazione dovuta alla funzione periodica di fix finger poiché ora ci

sono due finger table da mantenere.
2) Raddoppio della computazione dovuto al lookup poiché il lookup ora avviene consultando due finger table.

6.3.12 External Nodelist

Funzionamento

Questa è l'unica difesa non distribuita proposta in questa tesi. Essa infatti implica, benché minima, una gerarchia fra i nodi che partecipano all'overlay. Come avviene nelle reti peer-to-peer ad oggi più popolari, l'idea è quella di avere un set di server o supernodi fidati che conoscano, in ogni istante temporale, tutti i nodi o parte dei nodi presenti nell'overlay. Tali supernodi hanno lo scopo di costruire e di fornire periodicamente una Nodelist a tutti i nodi della rete. Questa Nodelist contiene un sottoinsieme random di nodi al momento presenti nell'overlay. Si legga il paragrafo sulla primitiva Nodelist per vedere come tale informazione di routing addizionale viene utilizzata, il suo mantenimento, e i vantaggi del suo impiego.
Qui di seguito abbiamo steso una proposta, senza definirne il protocollo, di come è possibile costruire e pubblicare la Nodelist ai nodi dell'overlay Chord. Nell'ipotesi di avere un solo supernodo che tiene traccia di tutti i nodi o una parte di essi, questo supernodo può estrarre uniformemente (in termini di chiave) i nodi per dar luogo ad una Nodelist uniforme sull'anello. Se da un lato la costruzione della Nodelist nel caso di un solo supernodo è semplice, dall'altro sappiamo che avere un solo supernodo non è scalabile con il numero di nodi e con il churn rate. La soluzione più scalabile è allora quella di impiegare una moltitudine di supernodi in cui ognuno ha in carico una parte dei nodi e di conseguenza solo una quota del churn totale. Il funzionamento di base è il seguente: quando un joining node n entra nella rete, estrae in modo random uniforme un supernodo s da una lista locale contenente tutti i supernodi. Il nodo n allora avvisa il suo supernodo s di essere entrato nell'overlay inviandogli direttamente, cioè senza impiegare l'overlay, un opportuno messaggio. Tale supernodo s sarà il supernodo che registra il nodo n inserendolo così nella sua Nodelist. Analogamente quando il nodo n esce avvisa il suo supernodo s che lo rimuove dalla sua Nodelist. Questo assegnamento nodi-supernodi non è legato alle posizioni nell'anello, infatti ogni supernodo si ritrova a tener traccia di una percentuale uniformemente distribuita sull'anello. Ciò significa che ogni supernodo ha una visione sì parziale (non registra tutti i nodi) ma globale della rete perchè registra nodi uniformemente distribuiti sull'anello. Inoltre poiché ogni singolo nodo seleziona il supernodo a cui si registra estraendolo in modo uniforme, questo garantisce che

in media i supernodi devono gestire una quota di N uguale tra loro e garantisce che in media i nodi ricevono una Nodelist di uguali dimensioni. Questo conferisce, in media, una fairness ideale sia tra i supernodi che tra i nodi.
Si osservi come il supernodo s, costruendo già da sé una Nodelist che copre l'intero anello, non ha bisogno di scambiare informazioni sui nodi con gli altri supernodi. Ciò sarebbe necessario se volessimo costruire un nodelist che cerchi di tener traccia dell'intera rete, ma questo chiaramente non è scalabile e ha senso se si intende utilizzare Chord.
Il vantaggio di impiegare la External Nodelist come supporto al lookup è presto spiegato. Le informazioni locali di Chord sono affette da inquinamento attivo dovuto alle informazioni false elargite dal subset maligno, perciò la frazione di entry maligne è ben superiore alla frazione f dei nodi maligni nell'overlay. L'External Nodelist al contrario, essendo un campionamento uniforme di nodi sull'anello, avrà anch'essa in media una frazione f di nodi maligni. Pertanto questo riduce drasticamente la probabilità che un lookup abbia come esito un next hop maligno.
E' utile sottolineare che, a parità di percentuale di nodi maligni, essi hanno soltanto un modo di incrementare la frazione maligna della Nodelist in modo che essa sia superiore alla frazione f di nodi maligni nella rete: impedire che i nodi onesti contattino i supernodi. Questo è possibile se i maligni hanno la possibilità di cancellare o corrompere le liste dei supernodi a disposizione dei nodi onesti (ad esempio tramite virus) ma questo è un problema di computer security per cui esula da questa tesi. I maligni possono invece introdurre "rumore" (nodi inesistenti) nelle nodelists dei supernodi. Lo possono fare ad esempio inviando periodicamente messaggi di join fasulli cioè relativi a nodi non realmente presenti nella rete, ad esempio creando socket random (da cui le relative chiavi). I nodi maligni sarebbero però costretti compiere IP spoofing su tali messaggi fasulli di join in quanto se usassero il loro indirizzo IP originale, si avrebbero entry multiple nella Nodelist che quindi non hanno alcun effetto, poiché non ha senso avere entry uguali. Allora, un metodo per scongiurare la possibilità dei maligni di introdurre questo rumore nelle Nodelist è il seguente: un supernodo a seguito della ricezione di un messaggio di join, prima di inserire nella propria Nodelist il nodo che ha inviato tale messaggio di join, può sfidare tale nodo attraverso il noto principio challenge-response: il supernodo invia al sedicente joining node un messaggio contente un numero random (challenge). Se il messaggio response non arriva al supernodo entro un timeout, oppure arriva con numero random differente, il supernodo non inserisce il nodo n nella sua Nodelist.

Overhead di traffico

L'overhead di traffico consiste in due contributi: l'invio da nodo a supernodo di un messaggio

di join e di leave e l'invio della Nodelist da supernodo a nodo. Un messaggio di join/leave costa ben poco in termini di banda del joining node n poiché è solo uno (solo al momento della join/leave) e non deve contenere nessun overlay layer payload. Lato supernodo se il churn rate totale della rete è C, ogni supernodo, data l'uniformità della selezione dei supernodi da parte di ogni joining node, è equamente ripartito fra gli S supernodi. Pertanto un supernodo riceve i messaggi di join e di leave con frequenza media C/S, quindi all'aumentare dei supernodi diminuisce il carico per supernodo ma d'altra parte ogni supernodo si ritrova ad avere Nodelist più corte a tutto svantaggio dei nodi inquinati dell'attacco Eclipse. Analizziamo quanto costa l'invio della Nodelist. Ogni entry della Nodelist al minimo costa un socket. Se siamo in una rete da 1000 nodi e ci sono S = 5 supernodi, ogni supernodo ha una nodelist contenente circa N/S = 1000/5 = 200 nodi e comporta quindi un payload di 200·socket_size = 1200 byte, dimensione che nella maggior parte dei casi non crea frammentazione IP. Se al contrario la dimensione di una Nodelist (rapporto N/S) è molto elevata, non è ragionevole trasmetterla per intero ed è meglio trasmettere un suo campionamento (uniforme nello spazio delle chiavi). Ovviamente il costo in termini di banda è linearmente proporzionale alla frequenza con cui il supernodo pubblica la Nodelist ai propri nodi (publication_rate). E' quindi ragionevole che tale frequenza sia aggiustabile in funzione del churn rate percepito dal supernodo, qui definito come frequenza di messaggi di join e di leave che il supernodo riceve. Ad esempio, il supernodo pubblica la propria Nodelist a tutti i nodi ad esso registrati ad ogni x modifiche (inserimenti + eliminazioni) della Nodelist dove x è un parametro arbitrario che può essere fissato tanto minore quanto è maggiore la banda disponibile. Se C/S è il churn rate a carico di un supernodo allora il publication rate della Nodelist è publication_rate = (C/S)/x. Ciò significa che la frequenza di messaggi di Nodelist trasmesse da un supernodo è nodelist_rate = (N/S)·((C/S)/x) dove N/S il numero di nodi di cui il nodo tiene traccia. Nel caso di IP multicast si avrebbe però un enorme vantaggio, infatti si avrebbe nodelist_rate = (C/S)/x il quale, coincide con la frequenza di ricezione delle Nodelist da parte dei nodi.

Overhead di memoria

Se ci sono N nodi e S supernodi allora la dimensione media della Nodelist di un supernodo è N/S. Se ad esempio abbiamo ben $N = 10^6$ nodi e S = 10 supernodi si ha N/S = 100000 entry (quindi 100000·6 = 600000 byte = 586 KB), il che è una memoria contenuta.

Overhead computazionale

I nodi devono inviare un messaggio di join e di leave e ricevere periodicamente le Nodelist. Inoltre il lookup viene effettuato anche sulla Nodelist oltre che sulla successor list e sulla

finger table.

6.4 Considerazioni aggiuntive

In questo paragrafo discutiamo le proposte di contromisura appena presentate, oltre ad approfondimenti alcuni dei quali non necessariamente legati alle contromisure qui trattate.

6.4.1 Confronto tra le contromisure

Nella tabella sottostante vi è una sintesi delle difese proposte in questa tesi in funzione delle loro caratteristiche.

Defense	Local	Distributed	Lookup	Reactive	Proactive	Optimization	Distance Test	Nodelist	Non Chord Messages
Alternative First Hop	X	X		X			X		
Redundant First Hop	X	X			X				
Delete Far Successors	X	X		X			X		
Augmented Fix Finger		X			X				X
Source Enhanced Local Info	X	X	X		X	X		X	
Path Enhanced Local Info	X	X	X		X	X		X	
Blacklist	X	X		X			X		X
Whitelist	X	X	X		X	X		X	X
Challenge Neighbor		X		X			X		
External Nodelist			X		X	X		X	X

Tab. 7: *Confronto fra le contromisure in funzione di diversi criteri.*

- Locale: una difesa è locale se non richiede lo scambio di messaggi estranei a quelli definiti dal protocollo Chord.
- Distribuita: una difesa è distribuita se non prevede una gerarchia di nodi. Ciò significa che tutti i nodi, nell'ambito di tale difesa, hanno esattamente lo stesso ruolo.
- Lookup: la difesa introduce una nuova struttura dati di lookup che si affianca alle strutture dati normali (finger table e successor list) per ottimizzare il lookup risultante da queste ultime.
- Reattiva: la difesa tenta di correggere azioni classificate non corrette, ossia non conformi alle regole protocollari. Questo approccio richiede prima un'identificazione dell'azione maligna e successivamente il tentativo di correzione.
- Proattiva: la difesa agisce indipendentemente dall'esito delle azioni previste da protocollo. Una soluzione di difesa proattiva ad un'azione x è infatti costituita soltanto dalla funzione di difesa, senza identificazione. L'algoritmo, attraverso la sua azione, cerca di ottenere dall'azione x il risultato corretto.

- Ottimizzazione: alcune difese sono di per sé delle ottimizzazioni sul lookup, infatti incrementando le informazioni locali di routing si migliora il routing, da intendersi come una riduzione dell'hop count medio dei path.
- Distance Test: la difesa si avvale della primitiva di difesa Distance Test.
- Nodelist: la difesa si avvale della primitiva di difesa Nodelist.
- Non Chord Message: la difesa necessita di messaggi non definiti nel protocollo Chord.

6.4.2 Discussione sulla primitiva Nodelist

La primitiva Nodelist viene utilizzata sempre nello stesso modo per effettuare i lookup, indipendentemente dalla difesa che la utilizza. Ciò che cambia da difesa a difesa è solo il modo in cui è alimentata la Nodelist, cioè quello che fa la differenza è quali nodi sono inseriti nella Nodelist, la quale ricordiamolo, può essere implementata come una struttura dati dinamica. Nel caso di Source Enhanced Routing Info i nodi che alimentano la Nodelist sono il nodo sorgente e il nodo precedente dei messaggi ricevuti, nel Path Enhanced Routing Info sono invece i nodi del path del messaggio ricevuto, nella Whitelist sono i nodi che sono stati raggiunti dal nodo in questione e che sono stati classificati onesti, in External Nodelist sono invece un campionamento random (ad es. uniforme) dell'overlay. Non stupisce che la versione più efficace e stabile di Nodelist è quella realizzata dalla External Nodelist perchè in quel caso la Nodelist stessa non dipende dalle dinamiche di traffico della rete, né tantomeno dall'effetto di partizionamento dei maligni. Inoltre, essendo la External Nodelist un campionamento random dei nodi, il suo grado di inquinamento medio coincide con la frazione maligna f dell'overlay, la quale è in genere assai inferiore dell'inquinamento dei nodi onesti. Le altre versioni di Nodelist invece sorgono dall'osservazione del traffico, il quale è fortemente condizionato dai maligni. Come si vedrà nei risultati delle simulazioni, questo spiega il fatto che l'External Nodelist è la difesa più efficace di quelle che impiegano la primitiva Nodelist. Non solo, l'External Nodelist è pure la migliore difesa singola, come vedremo nel paragrafo dei risultati.

6.4.3. Binary Chord

In [16] è discussa una modalità alternativa di assegnazione della chiave ad un nodo, chiamata Binary Chord. Gli autori di [16] hanno pensato tale meccanismo con lo scopo di rendere più uniforme le quantità di chiavi (archi di anello) di cui i nodi sono responsabili. In

altre parole, Binary Chord riduce la varianza della distanza fra nodi. Nel nostro contesto tale soluzione può aiutare a ridurre i falsi negativi e positivi in tutte le difese in cui si impiega la distanza media fra nodi (primitiva Distance Test). Nel modello tradizionale la chiave di un nodo è l'hash del suo IP address, e nella solita ipotesi di hash uniforme, la distanza media fra nodi è una variabile geometrica, la quale ha un coefficiente di variazione (standard_dev / mean) pari a 1. Con Binary Chord la chiave del nodo entrante è piuttosto il valore medio tra la chiave del suo predecessor e del suo successor dell'hash dell'IP address. Omettiamo la dimostrazione, ma questo implica che la distanza media fra nodi ha un coefficiente di variazione inferiore (0.66 contro 1) e quindi non è più geometrica. La minore dispersione delle distanze dovrebbe far diminuire la dispersione delle stime della distanza media fra nodi. Occorre però ridefinire la primitiva Distance Test. Supponiamo di essere nella solita situazione in cui un nodo s manda un messaggio di chiave k che termina su un nodo di chiave d. Il nodo s riceve allora la risposta (overlay ack / fix finger response) dal nodo d e da essa vuole verificare se il nodo d è veramente il successor della chiave k. Il problema è che il nodo s non può più applicare la Distance Test come è stata definita, infatti non conosce la chiave d in quanto essa non è più ricavabile dall'hash dell'indirizzo IP del nodo d. Soluzione possibile: il nodo s verifica se il nodo d che risponde è verosimilmente il successor del messaggio diretto di s o no calcolando l'hash H dell'IP address della risposta. Detta I = distanza_media_nodi e α un parametro reale libero, se l'hash di tale IP address cade fuori dall'intervallo (h-αI, h+αI) allora il nodo d è considerato troppo lontano per essere il successor della chiave k. Anche qui, per evitare che il Distance Test sia ingannato da IP spoofing, si possono adottare le verifiche sull'indirizzo IP di quando abbiamo parlato del Distance Test. Infine se la distanza fra nodi non è più geometrica allora E[distance_reponse] non è più pari a distanza_media_nodi = 2^m, perciò, se si vuole usare la primitiva di difesa Distance Test, occorre trovare un altra relazione fra response_distance e N.

6.4.4 Sicurezza delle chiavi dei nodi

Per evitare posizionamenti strategici nell'anello da parte dei nodi maligni, occorre limitare il più possibile la libertà di posizionamento di un nodo nell'anello. Pertanto la chiave di un nodo è meglio che sia l'hash del solo IP address che del socket, infatti è assai più agevole modificare il numero di porta che l'IP address e questo rende più fattibile una ricerca a forza bruta sull'hash per ottenere il socket corrispondente ad una chiave che cada in una certa regione dell'anello. In altre parole, se la chiave di un nodo fosse definita come l'hash del socket anziché del solo indirizzo IP, l'attaccante variando il numero di porta avrebbe a

disposizione un numero di posizioni possibili sull'anello che ha come limite superiore lo spazio delle porte (di solito 2^{16}).

Altri aspetti che riguardano la sicurezza del calcolo delle chiavi dei nodi sono:

1) Collisione hash: un nodo maligno può impersonare un nodi onesto n già esistenti (si autoassegna, se possibile, un IP address diverso da quello del nodo n ma tale che l'hash sia uguale alla chiave del nodo n). Pertanto la funzione hash deve possedere un adeguato livello di second preimage resistance.

2) Inversione hash: un nodo maligno può posizionarsi esattamente alla chiave desiderata. Pertanto l'hash deve possedere un adeguato livello di preimage resistance.

Ricordiamo che l'impersonamento di un maligno di un onesto o il posizionamento deciso dal maligno, possono essere utili per intercettare e/o deviare il traffico proveniente o indirizzato a specifici nodi vittima.

6.5 Stima della distanza media fra nodi

Abbiamo visto che molte difese si avvalgono della conoscenza della distanza media fra nodi, che avevamo indicato distanza_media_nodi. Ricordiamo queste difese: Previous Fix Finger Call, Delete Far Successors, Whitelist, Blacklist. Esse sono tutte le contromisure che invocano la primitiva Distance Test. Questa necessità fa sorgere un problema: un nodo infatti, non conosce a priori il valore di distanza_media_nodi a meno che non lo chieda ad un sistema centralizzato o semi-centralizzato che tenga traccia in tempo reale di tutti i nodi presenti nella rete. In questo caso infatti, detto N il numero di nodi nella rete in un certo istante, allora si ha: distanza_media_nodi = spazio_chiavi/N = 2^m/N. Ovviamente questo è vero qualunque sia la distribuzione della distanza_media_nodi. Un tale sistema però non scala bene con il numero di nodi N e soprattutto con il churn rate perchè va a sovraccaricare la funzione di tracciamento dei nodi da parte dei supernodi.
Per questa ragione abbiamo pensato ad un sistema di stima di distanza_media_nodi locale e per cui distribuito. Il principio di località è conferito dal fatto che tale sistema non richiede lo scambio di messaggi appositi fra i nodi. Un meccanismo locale non implica nessun overhead di banda ed è anche più affidabile perchè esente da perdite di messaggi, corruzione da parte

dei maligni, ecc. Il principio di località implica anche che il sistema proposto è distribuito, dal momento che non richiede la presenza di alcun supernodo. In ogni caso, è importante avere una certa precisione di stima di distanza_media_nodi. Quando infatti una stima è lontana dal valore vero, la classificazione delle risposte è poco affidabile a causa di una maggior percentuale di falsi positivi e falsi negativi. Per falsi positivi intendiamo l'esito falso di Distance Test quando in realtà è vero. L'idea di base è quella di effettuare la stima periodicamente, dal momento che a causa del churn, se varia il numero di nodi N varia pure la distanza media fra nodi consecutivi $2^m/N$. Consideriamo un generico nodo n che deve stimare distanza_media_nodi in un certo istante. La nostra stima fa leva sulla definizione di successor list. Infatti, dal momento che le entry della successor list sono nodi consecutivi è automatico fare la media delle succlist_size distanze fra le succlist_size entry della successor list, più il nodo corrente. Siccome tuttavia la successor list può essere inquinata dai nodi maligni è opportuno escludere dalla stima le distanze fra entry maligne poiché esse in generale non sono nodi consecutivi bensì nodi sparsi sull'anello. Le entry maligne quindi introducono un bias per eccesso nella stima di distanza_media_nodi poiché le loro distanze in media sono 1/f > 1 volte maggiori di quelle fra successori. Per effettuare questa stima numerica, vi sono due fattori da considerare:

1) Il numero di campioni disponibili
2) La capacità di reiettare i campioni outlier

Per quanto riguarda il fattore 1 se maggiore è la lunghezza della successor list, allora si ha a disposizione un maggior numero di campioni per calcolare la media campionaria della distanza fra nodi consecutivi. Come sappiamo la lunghezza della successor list (succlist_size) è un parametro libero, a differenza della finger table che ha lunghezza massima pari a m entry. Se le entry delle successor list sono corrette – e la condizione necessaria è che non vi sia attacco Eclipse – la stima della distanza_media_nodi è migliore poiché non vi sono outliers. Tuttavia, nel nostro caso il valore distanza_media_nodi è utile proprio quando c'è attacco, perciò quando nella successor list possono essere presenti entry non corrette che dunque danno origine ad outliers. Per questa ragione abbiamo pensato ad un sistema di "outlier rejection" che tenta di individuare gli outliers, ovvero le distanze che non sono i campioni voluti, cioè che non sono distanze fra nodi adiacenti. Ricordiamo che nel nostro caso gli outlier sono soltanto le distanze fra nodi maligni e quindi in generale non consecutivi, perciò sono valori la cui media è pari a (1/f)·distanza_media_nodi dove f è la frazione di nodi maligni. Dato che 1/f > 1, di conseguenza gli outlier sono valori evidentemente più elevati. Se f = 0.05 (5% nodi maligni) allora gli outlier hanno valore medio

1/0.05 = 20 volte la distanza_media_nodi. Il sistema proposto di stima, che punta sulla robustezza agli outlier, si basa sulla constatazione che le entry maligne, poiché in generale sono le più lontane, si concentrano nella parte finale della successor list. Quindi se ad esempio la entry successor(k) è maligna allora le entry successive sono tutte maligne poiché successor(k) è un nodo maligno e di conseguenza punta solo ad altri maligni che stanno davanti a lui in senso orario. Fatte queste considerazioni, e ricordando che i campioni sono le distanze fra entry consecutive della successor list, il sistema di stima evolve nelle seguenti fasi:

1) Calcolo della stima al tempo presente

Per semplicità di notazione indichiamo con la lettera “e” la stima della distanza media fra nodi (estimated_distance_node_mean). Con il simbolo “successor(0)” ci riferiamo alla prima entry della successor list, quindi il successor del nodo corrente che effettua la stima.

```
e = successor(0) – current_node      mod 2^m

for (k = 1; k < succlist_size; k++) {

      interval(k) = successor(k) – successor(k-1)     mod 2^m
      if (interval(k) < p·e)
            e = (k/(k+1))·e + (1/(k+1))·interval(k);
      else
            break;
}
```

Fig. 37: *Pseudocodice dell’algoritmo di stima del valore corrente della distanza internodo media.*

Il numero reale estimated_distance_node_mean viene poi arrotondato all'intero più vicino dato che lo spazio delle chiavi è un insieme di interi. Quello che fa l’algoritmo sopra esposto è quello di calcolare progressivamente la stima corrente di distanza_media_nodi includendo ad ogni nuova iterazione un campione in più per la stima e si interrompe quando si incontra il primo campione che è ritenuto outlier. Un campione è ritenuto outlier quando è maggiore della soglia dell'iterazione corrente p·estimated_distance_nodes_mean. L'output è quindi l'ultima stima di distanza_media_nodi, ossia il valore ottenuto nell'ultima iterazione. Se in nessuna iterazione il rispettivo campione è classificato outlier, la stima avviene impiegando

tutta la successor list, ossia calcolando le distanze internodo fino alla fine della successor list. Il valore reale p è un parametro da regolare in modo che la stima sia la più accurata possibile: minore è p e maggiore è la capacità di reiezione degli outlier ma cresce la probabilità di escludere campioni validi, mentre se p è elevato diminuisce la reiezione degli outlier ma si considerano più campioni validi.

2) Inserimento della nuova stima in una sliding window l[i]. Questa sliding window contiene le ultime L - 1 stime più quella corrente (in totale L stime). Comportandosi come una coda FIFO, a regime la sliding window è piena cioè contiene L stime, quindi l'inserimento della stima più recente implica l'eliminazione di quella più vecchia.

3) Calcolo della stima come media aritmetica (oppure pesata) delle W stime contenute nella sliding window. La ragione per cui abbiamo pensato di calcolare la stima corrente come media mobile è quella di coinvolgere un maggior numero di campioni usati per la stima. Infatti, includendo anche le stime precedenti, equivale a tener conto anche delle distanze misurate in precedenza (i campioni) e pertanto si allarga il supporto statistico sul quale è effettuata la stima. Più il numero di nodi N è costante durante lungo la finestra temporale coperta dalla sliding window e maggior è il numero di campioni coerenti ossia che si riferiscono allo stesso N e migliore è la stima dato il maggiore supporto statistico. E' chiaro tuttavia che è necessario ridurre la “retro-estensione” della sliding window all'aumentare della variazione di N per evitare di includere campioni inconsistenti. Ciò dipende dall'applicazione specifica e se infatti si prevede un alto dinamismo nella rete (veloci variazioni di N) è bene accorciare il periodo di calcolo delle stime e/o ridurre l'estensione L della sliding window in numero di stime.

In conclusione i pregi di questo meccanismo di stima è la relativa semplicità di implementazione, non è troppo oneroso dal punto di vista computazionale e non richiede la preparazione di un training set come invece avverrebbe con un approccio di stima tramite data mining. Per contro ha uno svantaggio operativo considerevole: il valore da assegnare al parametro reale p. Questo di per sé è un training. Attraverso parecchie simulazioni abbiamo dimostrato che, con un opportuno tuning di p, è possibile ottenere valori di stima i quali, mediati nel tempo con il meccanismo sliding window, sono abbastanza prossimi al valore vero di distanza_media_nodi ossia $2^m/N$. In ogni caso è bene osservare che non ci occorre una grandissima precisione di stima perchè comunque, anche se queste stime sono polarizzate per eccesso a causa degli outlier, si può sempre ridurre il valore reale factor che determina la soglia T = factor·stima_distanza_media_nodi. Diminuendo factor la soglia è più

restrittiva quindi si riducono i falsi negativi ma aumentano i falsi positivi, e viceversa aumentando factor. Il vero problema quindi non è la polarizzazione dello stimatore, piuttosto è la sua varianza delle diverse stime, le quali cambiano da nodo a nodo e, in caso di churn, anche nel tempo. Negli istogrammi delle frequenza statistica degli errori avremo modo di valutare la varianza dall'errore. Quantifichiamo ora le probabilità di falsi positivi e negativi dal punto di vista analitico. Sappiamo che la distanza fra nodi consecutivi è una geometrica di media $2^m/N$ e che la distanza fra nodi maligni è una geometrica di media $2^m/fN$. Sia "negative" l'evento per cui la risposta viene dal successor di una chiave di destinazione (risposta vera) ed "positive" l'evento in cui la risposta viene da un nodo che non è successor della chiave di destinazione (risposta falsa). Inoltre per brevità indichiamo d = response_distance. Un nodo deve quindi tarare il fattore factor della soglia T = factor·distanza_media_nodi in modo da avere una certa probabilità di falso negativo e di falso positivo. Per fare questo è però necessario sapere i parametri p e p_f delle due geometriche in gioco, per cui è necessario conoscere N ed f che, in generale, non sono noti. Ricordiamo che la conoscenza di N equivale alla conoscenza di distanza_media_nodi, poiché differiscono solo di una costante nota: N = 2^m/distanza_media_nodi. Anche la frazione di maligni f non è nota a priori e comunque richiede la conoscenza del numero totale di nodi N. Pertanto se un nodo volesse regolare factor e per cui la soglia T in modo da avere determinati valori di falso positivo e negativo, occorre prima stimare la dimensione N dell'overlay e la frazione f maligna. In altre parole, il nodo deve stimare distanza_media_nodi = $2^m/N$ e distanza_media_nodi_maligni = $2^m/fN$. In questo ultimo paragrafo abbiamo proposto un metodo per stimare localmente distanza_media_nodi ma non distanza_media_nodi_maligni. Teniamo conto tuttavia che, parlando di stime e non di valori esatti, se si fissa T in modo da avere certe probabilità di falso positivo e negativo, di fatto non si hanno i valori voluti per tale calcolo parte da valori non esatti. In teoria è anche possibile avere una stima di distanza_media_nodi molto più accurata di quella che si può ottenere localmente, se però si impiega un protocollo opportuno fra i vari supernodi per tracciare il numero N dei nodi presenti.

La probabilità di falso positivo quando si verifica un negativo è:

$$P_{fp} = P(d > T \mid negative) = \sum_{i=T+1}^{2^m-1} p(1-p)^{i-1}$$

Dove p è il parametro della geometrica che rappresenta la distanza fra gli N nodi e dunque: p = $1/(2^m/N)$.

Invece la probabilità di falso negativo quando si verifica un positivo è:

$$P_{fn} = P(d < T \mid positivo) = \sum_{i=1}^{T-1} p_f \left(1 - p_f\right)^{i-1}$$

Dove p_f è il parametro della geometrica che rappresenta la distanza fra gli fN nodi maligni e dunque: $p_f = 1/(2^m/fN)$.

Tuttavia nelle nostre simulazioni abbiamo proceduto empiricamente nel tuning di factor, regolandolo sino a raggiungere le minime percentuale di messaggi catturati. Sia nelle simulazioni in cui abbiamo impiegato il valore esatto di distanza_media_nodi che quello stimato, i valori migliori di factor vanno indicativamente da 1 a 1.5.

Per ogni scenario di simulazione sperimentato (N = 100, 500, 1000 con f = 0.05 per i primi due e f = 0.02 per il terzo) consideriamo due errori:

a) Errore relativo assoluto

$$relative\_absolute\_error = \frac{|\ estimated - true|}{true}$$

b) Errore relativo

$$relative\_error = \frac{estimated - true}{true}$$

Come indice di bontà di stima mettiamo in evidenza il valore mediano dell'errore assoluto relativo. Abbiamo scelto la mediana anziché la media poiché quest'ultima risentirebbe di ogni outlier nella distribuzione (outlier di errore). Attenzione che prima per outlier intendevamo i campioni outlier per fare la stima, mentre per outlier di errore intendiamo i valori di errore di stima che sono molto lontani dalla mediana.

Nei grafici seguenti sono esposti gli istogrammi dell'errore relativo di stima e la mediana del valore assoluto dell'errore relativo di stima.

Risultati:

1) N = 100, f = 0.05, p = 5

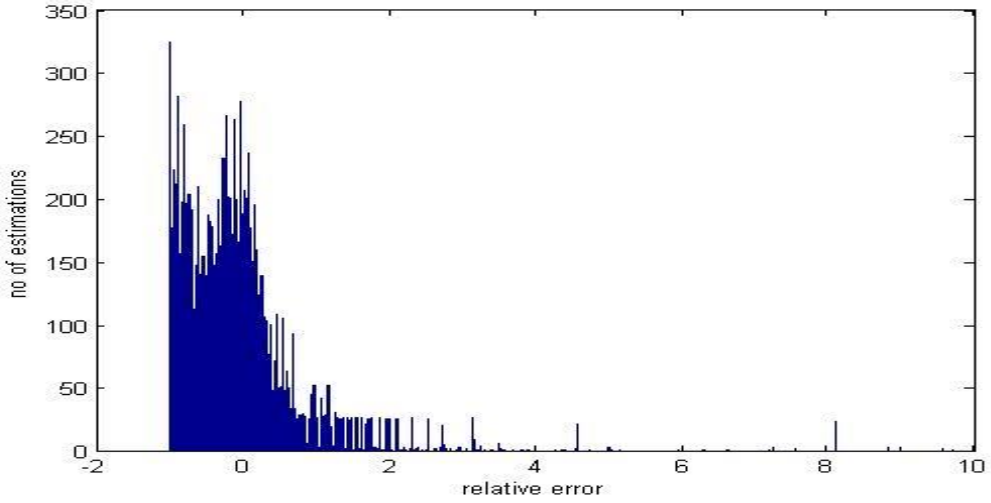

Fig. 38: *Distribuzione dell'errore relativo di stima della distanza media internodo.*

Mediana (errore relative assoluto) = 0.4

2) N = 500, f = 0.05, k = 5

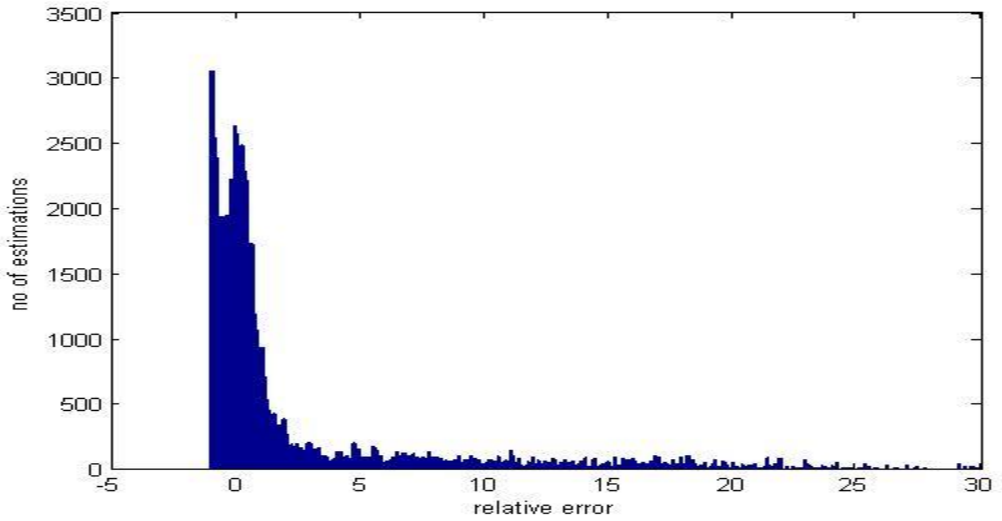

Fig. 39: *Distribuzione dell'errore relativo di stima della distanza media internodo.*

Mediana (errore relativo assoluto) = 0.74

3) N = 1000, f = 0.02, p = 5

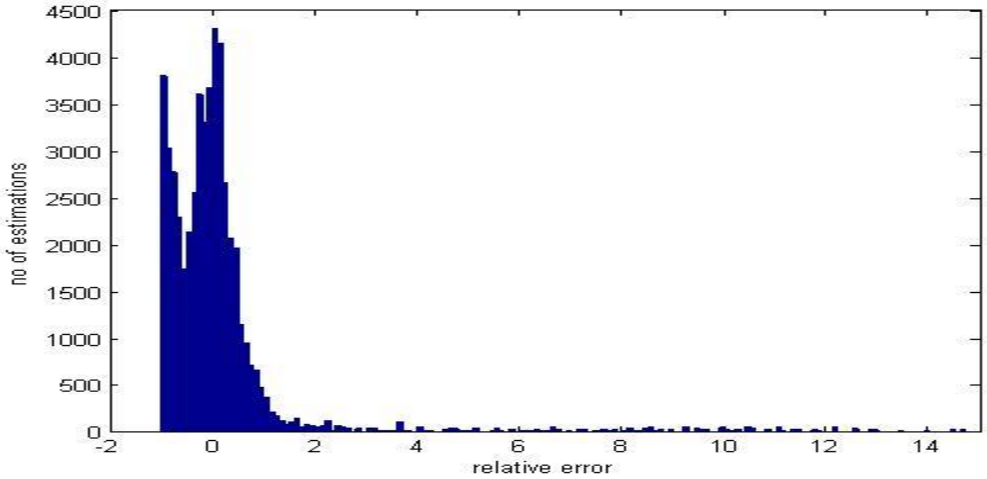

Fig. 40: *Distribuzione dell'errore relativo di stima della distanza media internodo.*

Mediana (errore relativo assoluto) = 0.33

Come si deduce dai tre grafici, l'algoritmo di stima è affetto da un errore positivo dovuto agli outliers delle distanza fra nodi maligni. In ogni caso, tuttavia, tale errore è molto concentrato attorno allo zero.

6.6 Risultati

Indichiamo con C la percentuale di messaggi catturati. Chiamiamo inoltre Ps la probabilità che un messaggio avente successor onesto termini su tale nodo onesto (probabilità di successo). La difesa ideale all'attacco Eclipse assicura Ps = 1, riducendo quindi l'attacco Eclipse al molto meno dannoso attacco Sybil. Se le difese sono in grado di assicurare Ps = 1, significa che la percentuale di messaggi catturati scende al livello di Sybil, ossia f·100. Di conseguenza il meglio che si può fare contro l'attacco Eclipse è f·100.

Se non diversamente specificato, i valori dei parametri di simulazione e del protocollo Chord comuni a tutte le simulazioni, sono riportati nel capitolo 4. Per quanto riguarda i parametri delle difese, si vedano le tabelle più avanti. In esse sono riportati i risultati di ogni difesa presa singolarmente. Per ogni difesa considerata (oltre al caso senza difesa) abbiamo considerato una rete da N = 100 nodi (con f = 0.05), una da N = 500 nodi (con f = 0.05) e da

N = 1000 nodi (con f = 0.02). Abbiamo scelto una frazione maligna inferiore per N = 1000 nodi poiché per grandi overlay è più verosimile che la frazione di nodi collusi sia inferiore. In tutti i casi poi abbiamo sempre tenuto lo stesso seed (per fairness di confronto). I parametri delle difese sono il valore factor (Distance Test) e %N (Nodelist). Nel caso in cui la Nodelist è preparata esternamente (External Nodelist) allora la Nodelist ha dimensione costante %N mentre nel caso in cui sia costruita localmente (Whitelist, Source / Path Enhanced Local Info) la Nodelist ha dimensione variabile e %N rappresenta il limite massimo imposto.

N = 100, f = 0.05	**Parameters**		**Results**			
Defense	factor	% N	% Captured	95% Conf	E[h]	E[hrel]
No attack	-	-	0	(0,0)	3,3	71
No defense	-	-	83	(78,86)	2	190
Alternative First Hop	1,2	-	57	(46,68)	2,5	131
Redundant First Hop	1,2	-	48	(40,55)	4	178
Delete Far Successors	1,2	-	60	(51,69)	1,2	150
Augmented Fix Finger	-	-	60	(55,65)	2,4	134
Source Enhanced Local Info	-	-	32	(22,42)	4,2	130
Path Enhanced Local Info	-	-	28	(21,36)	4	136
Blacklist	1,2	20	56	(42,70)	2,5	185
Whitelist	1,2	20	72	(67,77)	2	172
Challenge Neighbor	1,2	-	65	(52,76)	2,3	154
External Nodelist	-	20	17	(14,20)	2,9	114

Tab. 8: *Risultati delle difese singole con overlay da N = 100 nodi e frazione f = 0.05 di nodi maligni.*

N = 500, f = 0.05	**Parameters**		**Results**			
Defense	factor	% N	% Captured	95% Conf	E[h]	E[hrel]
No attack	-	-	0	(0,0)	4,4	67
No defense	-	-	85	(83,87)	2	248
Alternative First Hop	1,2	-	77	(74,80)	2,5	133
Redundant First Hop	1,2	-	72	(66,79)	2,9	146
Delete Far Successors	1,2	-	48	(44,52)	1,3	65
Augmented Fix Finger	-	-	75	(74,76)	2,4	120
Source Enhanced Local Info	-	-	74	(71,77)	3,6	158
Path Enhanced Local Info	-	-	45	(38,52)	4,5	91
Blacklist	1,2	20	69	(62,75)	2	95
Whitelist	1,2	20	75	(71,78)	2	121
Challenge Neighbor	1,2	-	78	(74,82)	2,4	134
External Nodelist	-	20	15	(14,16)	2,6	62

Tab. 9: *Risultati delle difese singole con overlay da N = 500 nodi e frazione f = 0.05 di nodi maligni.*

N = 1000, f = 0.02	Parameters		Results			
Defense	factor	% N	% Captured	95% Conf	E[h]	E[hrel]
No attack	-	-	0	(0,0)	4,9	64
No defense	-	-	87	(83,90)	2,3	224
Alternative First Hop	1,2	-	62	(44,79)	3,4	76
Redundant First Hop	1,2	-	50	(37,63)	3,7	78
Delete Far Successors	1,2	-	22	(11,31)	4,7	59
Augmented Fix Finger	-	-	49	(42,56)	3,5	82
Source Enhanced Local Info	-	-	79	(70,88)	5,6	234
Path Enhanced Local Info	-	-	33	(22,45)	5,6	100
Blacklist	1,2	20	48	(44,53)	3,3	67
Whitelist	1,2	20	57	(55,59)	2,3	74
Challenge Neighbor	1,2	-	27	(17,37)	4	64
External Nodelist	-	20	7	(5,9)	2,8	41

Tab. 10: *Risultati delle difese singole con overlay da N = 1000 nodi e frazione f = 0.02 di nodi maligni*.

Premettendo che i dati delle tab. 8, 9 e 10 si riferiscono ai parametri della simulazione, quindi per diversi valori dei parametri è possibile avere risultati peggiori o migliori, come si può notare dalle tabelle, le difese singole danno risultati migliorabili. Ad esempio con N = 100 nodi il meglio che si può fare con difesa singola è C attorno al 30% grazie alle informazioni di routing ottenute osservando i messaggi (Source/Path Enhanced Local Info). Come è giusto che sia, a parità di f se N è maggiore, C peggiora in quanto l'hop count medio dei path è maggiore e con essa la probabilità di cattura. In tutte e tre le tabelle le difese distribuite singole più o meno si equivalgono, tuttavia sembra che Path Enhanced Local Info sia la difesa distribuita singola più efficace. Non è difficile spiegare il perché: mentre tutte le altre difese distribuite (quindi External Nodelist esclusa) cercano di ripulire le proprie informazioni di routing a partire dalle stesse, le quali sono soggette all'inquinamento attivo, l'osservazione dei nodi sorgenti o lungo il path dei messaggi permettono una visione della rete meno distorta dall'effetto Eclipse. Questo principio si manifesta in tutta la sua chiarezza con la difesa non distribuita External Nodelist, la quale fornendo un campionamento dei nodi dell'overlay effettuato esternamente, quindi esente dall'inquinamento attivo, consente di ridurre drasticamente l'inoltro dei messaggi verso nodi maligni. Il prezzo da pagare per sfuggire dalla collusione dei nodi maligni è però quello di avere nodi esterni al Chord overlay che tengano traccia almeno di una parte dei nodi del Chord overlay. E' inoltre importante notare che le uniche due difese che si avvalgono delle sole informazioni locali di Chord (Alternative First Hop, Redundant First Hop) non danno risultati molto promettenti. Per esplorare fino a dove si può arrivare basandosi sulle sole informazioni locali di Chord,

sarebbe stato interessante valutare anche il routing ridondante, tuttavia per motivi di scalabilità non abbiamo potuto valutarlo e in ogni caso sarebbe una soluzione troppo costosa in termini di banda. Infine, oltre alla percentuale di messaggi catturati C, nelle tabelle valutiamo anche E[h] ossia l'hop count medio del path su tutti i messaggi terminati su nodi onesti e $E[h_{rel}]$ ossia l'hop count medio relativo su tutti i messaggi terminati su nodi onesti. Osservando E[h] si nota che il valore diminuisce in caso di attacco perchè i path che finiscono su nodi onesti sono più brevi, infatti più i path dovrebbero essere lunghi e maggiore è la probabilità che entrino nel subset maligno ed essi non sono conteggiati in E[h]. L'hop count relativo $E[h_{rel}]$ è invece un indicatore dell'efficienza del routing perchè è l'hop count normalizzato alla distanza (normalizzata) fra nodo sorgente e nodo destinazione. La citata distanza è normalizzata per non avere ordini di grandezza di $E[h_{rel}]$ troppo piccoli. Si consulti l'inizio di questo capitolo per vedere le formule di calcolo di E[h] e di $E[h_{rel}]$. Rispetto al routing corretto del protocollo (No attack), accade che $E[h_{rel}]$ peggiora (è più alto) mentre negli altri casi è paragonabile o addirittura migliora (è più basso).
Nella tabella seguente mostriamo, per ognuno dei tre scenari (N = 100, f = 0.05), (N = 500, f = 0.05) e (N = 1000, f = 0.02), le tre combinazioni migliori di difese distribuite e due di difese non distribuite. Per combinazione di difesa non distribuita intendiamo una combinazione contenente la proposta di difesa non distribuita (External Nodelist). Dato che abbiamo pensato una molteplicità di difese singole, è chiaro che il numero di potenziali combinazioni di difese è molto grande, perciò dopo aver sperimentato un largo numero di combinazioni di difese, siamo giunti alle cinque defense set proposte in legenda (vedi sotto).

Legenda difese:

A = External Nodelist
B = External Nodelist + Path Enhanced Local Info
C = Augmented Fix Finger + Whitelist + Path Enhanced Local Info
D = Augmented Fix Finger + Whitelist + Blacklist + Source Enhanced Local Info
E = Augmented Fix Finger + Delete Far Successors + Whitelist + Blacklist + Path Enhanced Local Info

Defense	N	Defense Parameters		Results			
		factor	% N	% Captured	95% Conf	E[h]	E[hrel]
No attack	100	-	-	0	(0,0)	3,3	71
No defence	100	-	-	83	(78,86)	2	190
A	100	-	20	17	(13,21)	2,3	122
B	100	-	20	10	(8,12)	1,9	109
C	100	1,2	20	13	(12,14)	3,3	117
D	100	1,2	20	16	(13,19)	3,3	170
E	100	1,2	20	12	(11,13)	2,8	141
No attack	500	-	-	0	(0,0)	4,4	67
No defence	500	-	-	85	(83,87)	2	248
A	500	-	20	14	(12,16)	2,6	60
B	500	-	20	10	(8,12)	2,2	49
C	500	1,2	20	17	(15,19)	2,5	56
D	500	1,2	20	16	(13,19)	2,8	60
E	500	1,2	20	18	(16,20)	2,5	54
No attack	1000	-	-	0	(0,0)	4,9	64
No defence	1000	-	-	87	(83,90)	2,3	224
A	1000	-	20	6	(5,7)	2,8	57
B	1000	-	20	4	(3,5)	2,5	46
C	1000	1,2	20	9	(8,10)	2,9	39
D	1000	1,2	20	8	(9,10)	3	38
E	1000	1,2	20	7	(5,9)	2,8	37

Tab. 11: *Risultati delle migliori combinazioni di difese nei tre scenari di simulazione.*

Nei tre istogrammi seguenti confrontiamo la percentuale di messaggi catturati dei cinque defense set scelti. In verde sono riportate le difese distribuite mentre in giallo quelle non distribuite.

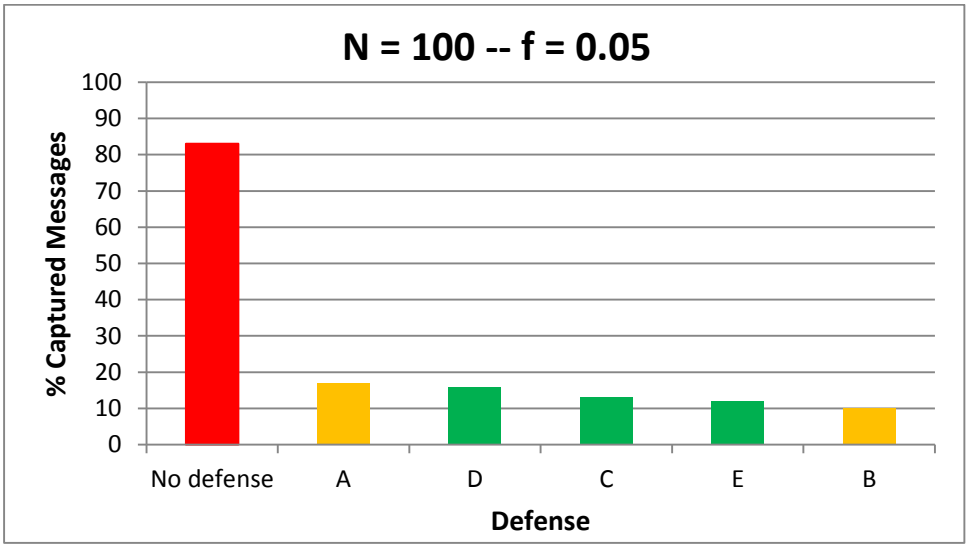

Fig. 41: *Percentuale messaggi catturati nel caso N = 100 nodi con frazione f = 0.05 maligna.*

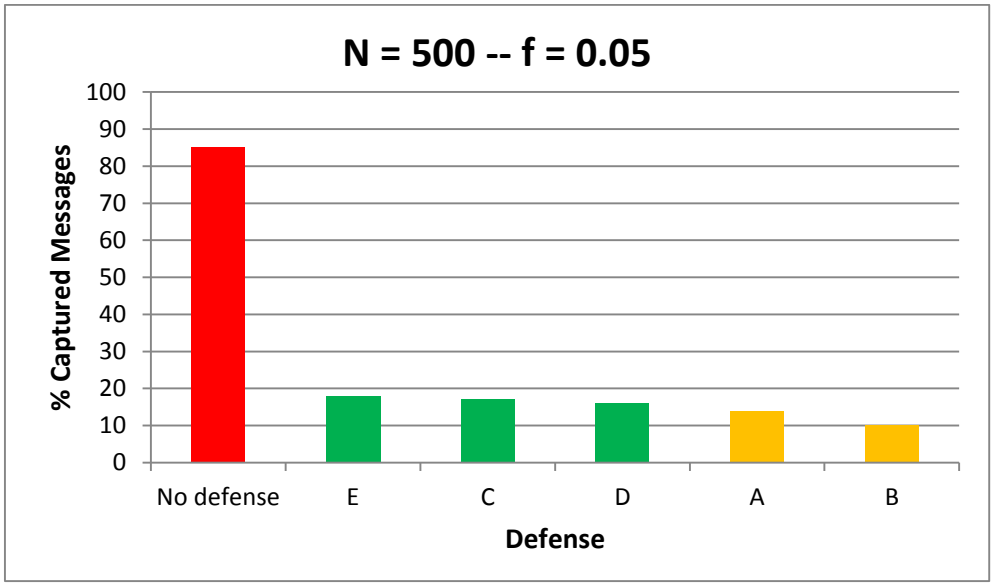

Fig. 42: *Percentuale messaggi catturati nel caso N = 500 nodi con frazione f = 0.05 maligna.*

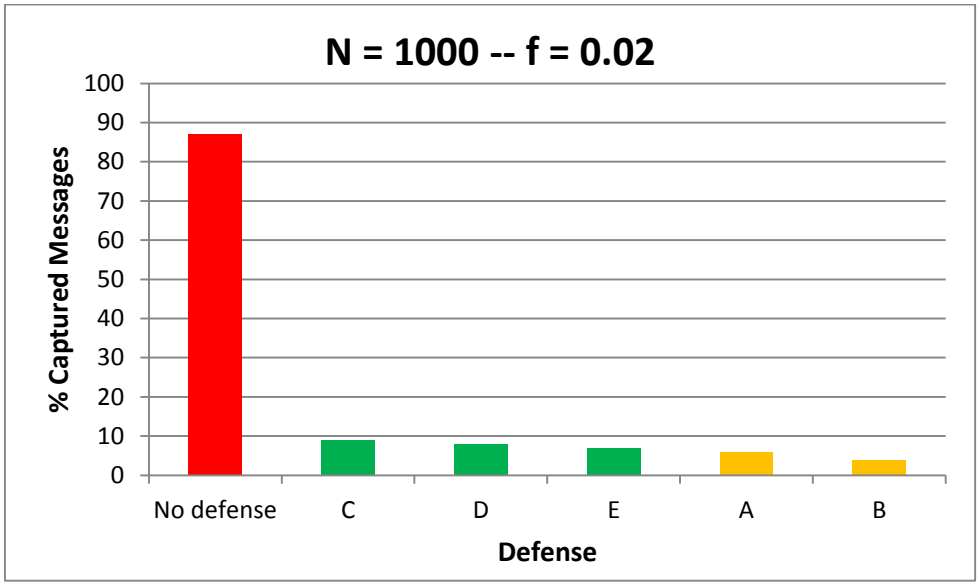

Fig. 43: *Percentuale messaggi catturati nel caso N = 1000 nodi con frazione f = 0.02 maligna.*

Dalla tab. 11 e da questi ultimi tre grafici si nota che:

1) A parità di parametri, le migliori difese distribuite si avvicinano molto a quelle non distribuite in termini di percentuale di messaggi catturati.

2) A parità di dimensione di Nodelist (20% di N), sia le difese distribuite che non distribuite, migliorano all'aumentare di N.

3) Ancor più delle difese singole, all'aumentare di N accade che $E[h_{rel}]$ scende sempre più sotto il valore che si ha nel Chord originale indifeso. Questo è segno che le difese fungono

anche da ottimizzatori di routing e il fatto che ciò sia più evidente all'aumentare di N, così come la riduzione della percentuale di messaggi catturati, è un'ottima cosa visto che Chord è stato pensato per applicazioni distribuite su larga scala.
Infine c'è un aspetto importante da tenere a mente: i grafici qui presenti si riferiscono ad una dimensione di Nodelist del 20%. Tuttavia, per reti piccole (fino a qualche migliaio di nodi) è fattibile avere dimensioni di Nodelist anche superiori al 20% di N, il che consentirebbe di ridurre ulteriormente la percentuale di messaggi catturati. Per tale motivo, in fig. 44 mostriamo come la sola difesa External Nodelist si comporta al variare della sua dimensione.

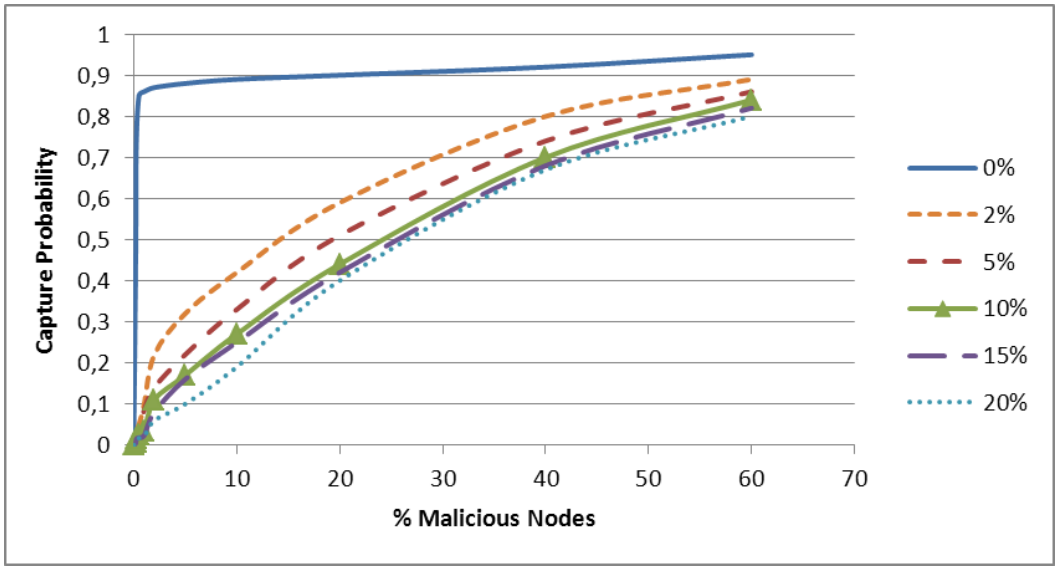

Fig. 44: *Probabilità di cattura di un messaggio per diverse dimensioni della External Nodelist.*

Come si nota, all'aumentare della dimensione della External Nodelist la curva di probabilità di cattura (percentuale di messaggi catturati/100) si abbassa. E' però curioso notare che anche con una Nodelist piccola (ad esempio 2%) si ha una forte decrescita dei messaggi catturati. Il motivo è che l'utilità dei nodi memorizzati nella External Nodelist sta nell'aiutare a sorpassare in senso orario i nodi eclissanti che creano le partizioni nell'anello. Di conseguenza, bastano pochi nodi sparsi uniformemente sull'anello per far sì che le maggior parte dei messaggi inoltrati riescano ad entrare nelle partizioni di anello in cui giacciono i successor onesti di tali messaggi. Il fatto che basti una piccola percentuale di Nodelist per avere un evidente effetto di contrasto alle eclissi, rende la Nodelist una soluzione scalabile in N. Ad esempio per $N = 10^6$ nodi un 2% di Nodelist, dato che una entry è il Chord socket di un nodo, richiede appena $20000 \cdot 6 = 117$ KB.

7
CONCLUSIONI

Dopo aver implementato l'attacco Eclipse al protocollo Chord, il presente lavoro di tesi è consistito nel proporre possibili soluzioni per identificare e per contrastare l'attacco Eclipse in una rete overlay Chord.

Per quanto riguarda il monitoraggio dell'attacco abbiamo presentato una soluzione di supervised machine learning basata su alberi decisionali di classificazione. E' emerso che variando la composizione del training set di addestramento dell'albero è possibile ottenere alberi con performance diverse nei diversi scenari di rete, specialmente al variare del numero di nodi. In conclusione abbiamo verificato che è fattibile identificare l'attacco Eclipse in Chord con elevata accuratezza quando il numero di nodi non è troppo distante dal numero di nodi degli scenari con i quali è stato addestrato l'albero.

Per quanto riguarda invece le difese, abbiamo osservato come le contromisure che cercano di sfruttare al meglio le sole informazioni di routing previste da Chord non possono dare risultati soddisfacenti data l'intrinseca vulnerabilità di Chord all'attacco Eclipse. Risultati apprezzabili, pertanto, si ottengono introducendo meccanismi ausiliari di ottenimento delle informazioni di routing. Lo scopo di tali meccanismi è quello di ridurre la frazione di entry maligne, mediante l'inserimento delle sole entry che più aderiscono alle regole del protocollo. Abbiamo anche dimostrato che è possibile avere una buona difesa mantenendo il sistema completamente distribuito, tuttavia prestazioni ancora migliori si ottengono con l'aiuto di supernodi fidati esterni.
Come ulteriore passo verso difese più efficaci, riteniamo interessante il routing Chord antiorario che, congiuntamente a quello tradizionale orario e alla External Nodelist, consentirebbe ai messaggi di raggiungere meglio le partizioni dell'anello causate dai nodi maligni.

Elenco figure

Elenco tabelle

Bibliografia

[1] G. Urdaneta, G. Pierre, M. Van Steen. A survey of DHT security techniques. *ACM Computing Surveys (CSUR), 43(2), January 2011.*

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications. *IEEE/ACM Trans. on Networking, vol. 11, no. 1, pp. 17–32, 2003.*

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker. CAN: A Scalable Content-Addressable Network. *In Proceedings of ACM SIGCOMM, 2001.*

[4] A. Rowstron, P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329–350, 2001.*

[5] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, J. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on selected areas in communications, Vol. 22, No. 1, January 2004.*

[6] P. Maymounkov, D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. *In Proc. 1st International Workshop on Peer-to-Peer Systems (Cambridge, MA). Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, Berlin, 2002.*

[7] E. Sit, R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. *In Proc. 1st International Workshop on Peer-to-Peer Systems. Cambridge, MA, Lecture Notes on Computer Science, vol. 2429. Springer-Verlag, Berlin, 2002.*

[8] K. Hildrum, J. Kubiatowicz. Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-Peer Networks*. In Proc. 17th International Symposium on Distributed Computing. Sorrento, Italy, Lecture Notes on Computer Science, vol. 2848. Springer-Verlag, Berlin, 2003.*

[9] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, D.S. Wallach. Secure Routing

for Structured Peer-to-Peer Overlay Networks. *In Proc. 5th Symposium on Operating System Design and Implementation (Boston, MA). ACM Press, New York, NY, 2002.*

[10] A. Singh, T.W. Ngan, P. Druschel, D.S. Wallach. Eclipse Attacks on Overlay Networks: Threats and Defenses. *In Proc. 25th INFOCOM Conference (Barcelona, Spain). IEEE Computer Society Press, Los Alamitos, CA., 2006.*

[11] B. Awerbuch, C. Scheideler. Towards Scalable and Robust Overlay Networks*. In Proc. 6th International Workshop on Peer-to-Peer Systems. Bellevue, WA, 2007.*

[12] T. Condie, V. Kacholia, S. Sankararaman, J. Hellerstein, and P. Maniatis. Induced Churn as Shelter from Routing Table Poisoning. *In Proc. of NDSS, February 2006.*

[13] D. Korzun, B. Nechaev, A. Gurtov. CR-Chord: Improving Lookup Availability in the Presence of Malicious DHT Nodes. *HIIT Technical Reports 2008-2, Dec. 2008.*

[14] Quinlan, J. R. C4.5: Programs for Machine Learning. *Morgan Kaufmann Publishers, 1993.*

[15] A. Binzenhofer, D. Staehle, and R. Henjes. On the Fly Estimation of the Peer Population in a Chord based P2P System. *In 19th International Teletraffic Congress (ITC19), Beijing, China, September 2005.*

[16] J. Cichon, A. Jasinki, R. Kapelko, M. Zawanda. How to improve the reliability of Chord? *Theoretical Aspects and Models of Large, Complex and Open Information Networks, ISI Foundation, Villa Gualino, Torino, Italy, November 19th - 21st, 2007.*