Il problema dell'autenticazione negli algoritmi di accordo in presenza di fallimenti bizantini

Federico Cicchi   Luca Cinti   Lorenzo Masetti

3 luglio 2001

La relazione affronta il problema dell'autenticazione in presenza di fallimenti bizantini. Il primo capitolo è un'inroduzione al problema dei generali bizantini che fu introdotto da Lamport, Shostack e Pease [1].

Il secondo capitolo descrive i problemi che si incontrano nell'implementazione di un sistema di autenticazione. Questi problemi sono ampiamente trattati in [2]; la sezione riguardante i freshness identifiers è basata su [4].

Il terzo capitolo descrive l'implementazione di un protocollo che realizza l'atomic broadcast in presenza di fallimenti bizantini. Questo protocollo è descritto in [3].

La bibliografia non comprende solo gli articoli che abbiamo consultato per scrivere la relazione ma anche altri articoli che abbiamo solo citato.

Indice

Capitolo 1

Introduzione al problema dei Generali Bizantini

1.1 Introduzione

Un sistema distribuito deve essere in grado di coesistere con il fallimento di uno o più dei suoi componenti. Un componente che fallisce potrebbe spedire informazioni conflittuali a parti diverse del sistema. Il problema della coesistenza con questo tipo di fallimenti è chiamato **Problema dei Generali Bizantini**.

Si immaginano diverse divisioni dell'esercito bizantino accampate al di fuori di una città nemica, ognuna delle quali con il proprio generale. I generali possono comunicare tra loro soltanto attraverso dei messaggeri e dopo aver osservato il nemico devono decidere un piano di attacco comune. Tuttavia, alcuni dei generali possono tradire, provando ad impedire ai generali fedeli di trovare un accordo. I generali devono quindi aver un algoritmo che garantisca che:

A. Tutti i generali decidano per lo stesso piano di azione.

I generali fedeli faranno quello che prevede l'algoritmo, ma i generali traditori possono fare quello che vogliono, per cui l'algoritmo deve garantire la condizione A a prescindere da quello che faranno i traditori. I generali fedeli non solo devono trovare un accordo, ma ne devono trovare uno che sia ragionevole, perciò si deve anche assicurare che

B. Un piccolo numero di traditori non possono causare l'adozione da parte dei generali fedeli di un cattivo piano.

Ogni generale osserva il nemico e comunica la sua osservazione agli altri. Sia $v(i)$ l'informazione comunicata dall' i-esimo comandante. Ogni generale usa un metodo per combinare i valori $v(1) \dots v(n)$ in un singolo piano di azione, con n numero di generali. La condizione A è assicurata se tutti i generali usano lo stesso metodo di combinazione delle informazioni, e la condizione B è raggiunta se si usa un metodo robusto. Ad esempio, nel caso in cui la sola decisione da prendere sia di attaccare o meno e $v(i)$ è l'opzione che l'i-esimo generale ritiene migliore, la decisione finale viene presa in base alla maggioranza dei voti. Un piccolo numero di traditori può influenzare la decisione se le opinioni dei generali fedeli sono ugualmente distribuite tra le due possibilità, nel qual caso nessuna delle due scelte può essere considerata completamente sbagliata.

Assumiamo di avere un metodo con il quale i generali comunicano i loro valori $v(i)$ a tutti gli altri. Il metodo più ovvio sarebbe quello di far si che ogni generale spedisca ad ognuno degli altri generali un messaggero che porta la propria valutazione. Tuttavia questo metodo non funziona perché la soddisfazione della condizione A richiede che ogni generale fedele ottenga gli stessi valori $v(1) \ldots v(n)$, mentre un generale traditore potrebbe spedire diversi valori a differenti generali. Perché la condizione A sia soffisfatta deve essere vero che

1. Ogni generale fedele deve avere le stesse informazioni

La condizione 1 implica che un generale non può necessariamente utilizzare un valore $v(i)$ ottenuto direttamente dall'i-esimo generale, poiché un traditore potrebbe spedire il proprio valore $v(i)$ diverso a diversi generali. Questo significa che se non si sta attenti ad assicurare la condizione 1 si potrebbe introdurre la possibilità che i generali basino la propria decisione su dati differenti, potendo quindi giungere a conclusioni diverse e incompatibili. Quindi introduciamo la seguente richiesta:

2. Se l'i-esimo generale è leale, allora il valore che spedisce deve essere usato da ogni altro generale come il valore $v(i)$.

Possiamo riscrivere la condizione 1 per ogni generale i (sia esso leale o traditore):

1'. Due generali leali usano gli stessi valori di $v(i)$

Le condizioni 1' e 2 sono entrambe condizioni sul singolo valore spedito dall'i-esimo generale. Si può quindi restringere la considerazione al modo in cui un singolo generale spedisce il suo valore agli altri; ciò si formula tramite messaggi spediti dai generali comandanti ai propri luogotenenti, ottenendo il problema dei Generali Bizantini.

Problema dei generali bizantini. Un comandante generale deve spedire un ordine ai suoi $n-1$ tenenti in modo che

IC1. Tutti i tenenti fedeli obbediscono allo stesso ordine.

IC2. Se il comandante generale è fedele, tutti i tenenti obbediscono al comando che ricevono.

Le condizioni *IC1* e *IC2* sono chiamate condizioni di consistenza interattiva. Si può notare che se il generale comandante è fedele, *IC1* segue da *IC2*. Comunque nessuno ci assicura che il comandante sia fedele.

1.2 Risultati di impossibilità

Apparentemente il problema dei generali bizantini può sembrare di facile soluzione, ma in realtà il fatto sorprendente è che se i generali possono spedire solo messaggi non autenticati è sufficiente che $\frac{1}{3}$ dei generali tradisca per far si che non ci sia modo di stabilire un piano comune, cioè non c'è modo di trovare un accordo efficiente tra i generali. Quest'ultima affermazione si può dimostrare

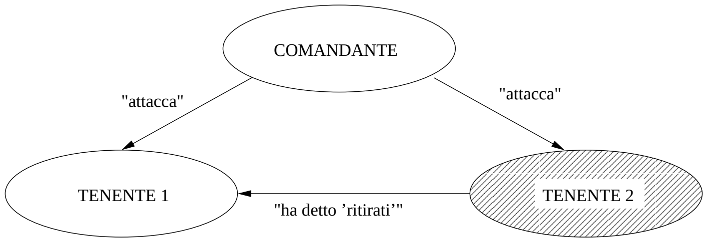

Figura 1.1: Il traditore è il Tenente 2

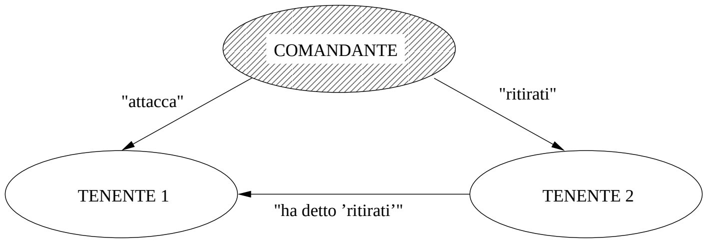

Figura 1.2: Il traditore è il comandante

formalmente, ma qui ci limiteremo al caso in cui ci sia un generale e due tenenti , uno dei quali tradisce, considerando per semplicità il caso in cui la sola decisione da prendere sia quella di attaccare o di ritirarsi.

Gli scenari possibili sono mostrati in figura 1.1 e 1.2.

Nel primo caso il comandante prende la decisione di attaccare e la comunica al tenente 2, che tradisce e spedisce al tenente 1 l'informazione scorretta di ritirarsi. Nel secondo caso il comandante traditore comunica al tenente 1 di attaccare ed al tenente 2 di ritirarsi. Quest'ultimo riporta fedelmente il comando di ritirarsi al tenente 1. In entrambi i casi il tenente 1 ottiene due messaggi opposti, ma non sa stabilire chi tra il comandante e il tenente abbia tradito, per cui non può eseguire con certezza nessuno dei due ordini, ed in ogni caso rischia che la decisione presa possa essere quella scorretta, mandando l'esercito o la propria divisione incontro ad una sconfitta sicura.

1.3 Una soluzione con messaggi firmati

Come mostrato dalle precedenti prove di impossibilità la difficoltà della risoluzione dei problemi dei generali bizantini, è anche dovuta all'abilità dei traditori, che possono far in modo di non farsi individuare dagli altri generali. Per restringere questa abilità è possibile far si che i generali si spediscano messaggi firmati in modo di avere la certezza del mittente di un messaggio. In pratica si effettuano quattro assunzioni, di cui le prime tre sono comuni alla soluzione con messaggi non criptati:

A1 Ogni messaggio spedito è ricevuto correttamente (assumiamo il canale di comunicazione sicuro)

A2 Chi riceve un messaggio sa chi lo ha spedito

A3 L'assenza di messaggi può essere rilevata

A4 **(a)** La firma di un generale fedele non può essere contraffatta, e qualsiasi alterazione del contenuto della sua firma può essere rilevata.

(b) Chiunque può verificare l'autenticità della firma di un generale.

Si può notare che non si fa nessuna assunzione circa la firma dei generali traditori.

Con l'introduzione dei messaggi firmati, i risultati di impossibilità che riguardavano la soluzione con messaggi orali, non sussistono più ed infatti si può mostrare un algoritmo che riesce a coesistere con m traditori, per un qualsiasi numero di generali (Il problema ha senso comunque se i generali sono almeno $m + 2$).

In tale algoritmo, il comandante spedisce un ordine firmato ad ognuno dei suoi tenenti. Ogni tenente aggiunge la sua firma e spedisce l'ordine agli altri tenenti. Si assume di avere una funzione *choice* che applicata ad un insieme di ordini ne restituisce uno. Le sole richieste su questa funzione sono:

1. Se l'insieme V consiste di un solo comando v, $choice(V) = v$
2. $choice(\{\}) = ritirati$

Una possibile scelta per *choice* potrebbe essere che prenda come ordine l'elemento mediano di V, assumendo un ordinamento degli elementi di V. Nell'algoritmo denoteremo con *x:i* il valore x firmato dal generale i. Quindi, *v:j:i* denota il valore v firmato da j, e il valore *v:j* firmato da i. Assumeremo che il generale 0 sia il comandante. Ogni tenente i mantiene un insieme di ordini V_i, contenente l'insieme di ordini correttamente firmati (se il comandante è leale, questo insieme non dovrebbe contenere mai più di un ordine). È importante non confondere l'insieme V_i degli ordini ricevuti, con l'insieme dei messaggi ricevuti.

Algoritmo $SM(m)$
Inizialmente $V_i = \{\}$

(1) Il comandante firma e manda il suo ordine a tutti i tenenti

(2) Per ogni i

(A) Se il tenente i riceve un messaggio della forma $v : 0$ dal comandante e non ha ancora ricevuto altri ordini

(i) pone $V_i = \{v\}$

(ii) Spedisce il messaggio *v:0:i* ad ogni altro tenente.

(B) Se il tenente i riceve un messaggio della forma $v{:}0{:}j_1 : \ldots : j_k$ e v non fa parte del suo insieme V_i

(i) aggiunge v a V_i

(ii) se $k < m$, spedisce il messaggio $v{:}0{:}j_1 : \ldots :j_k :i$ ad ogni altro tenente diverso da $j_1 \ldots j_k$.

(3) Per ogni i: quando l'i-esimo tenente non riceve più messaggi, obbedisce all'ordine $choice(V_i)$.

Il parametro m indica quindi il numero di round in cui si svolge l'algoritmo: se indichiamo con r il numero di round si ha che $r = m + 1$. Ad esempio se $m = 1$ l'algoritmo si svolge in due round: nel primo il comandante comunica il suo valore a tutti i tenenti, nel secondo i tenenti si scambiano i valori ricevuti per controllare se il generale ha tradito.

Da notare che nel passo (2), il tenente i ignora qualsiasi messaggio contenente l'ordine v che è già nell'insieme V_i

Ora dobbiamo mostrare come un tenente determina al passo (3) che non riceverà più messaggi. Per induzione su k, è facile mostrare che per ogni sequenza di tenenti $j_1, \ldots, j_k$ con $k \leq m$, un tenente può ricevere al più un solo messaggio della forma $v : 0 : j_1, \ldots, j_k$ al passo (2). Se richiediamo che il tentente j_k mandi il messaggio o altrimenti mandi un messaggio riportando che non spedirà quel messaggio, allora sarà facile decidere quando tutti i messaggi sono stati ricevuti. In alternativa è possibile utilizzare un time-out, per determinare quando la spedizione dei messaggi è terminata.

Si può notare che al passo (2), il tenente ignora qualsiasi messaggio che non ha l'appropriata forma di un valore seguito da una stringa di firme. Se pacchetti di identici messaggi sono usati per permettere di avere copie dei messaggi, significa che si perdono tutti quei pacchetti che non hanno un sufficiente numero di identiche, appropriate messaggi firmati. (Dovrebbero essere $(n-k-2)(n-k-3)\ldots(n-m-2)$ copie del messaggio che è stato firmato da k tententi).

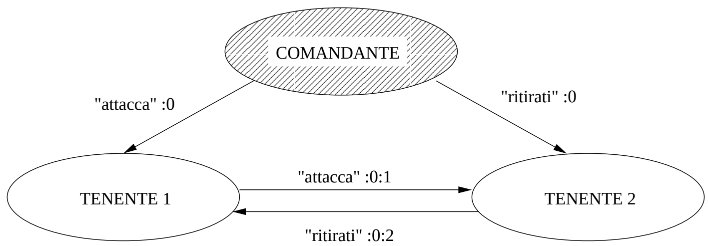

Figura 1.3: Algoritmo SM(1). Il traditore è il comandante

La Figura 1.3 illustra l'evolversi dell'Algoritmo SM(1) nel caso di tre generali quando il comandante è un traditore. Il comandante manda un messaggio di attacco ad un tenente e di ritirata all'altro.

Entrambi i tenenti ricevono i due ordini al passo (2), così dopo il passo (2) $V_1 = V_2 = \{"attacco", "ritirata"\}$, ed entrambi obbediscono all'ordine $choice(\{$ "attacco","ritirata"$\})$. Qui, contrariamente a quanto accade nella situazione analoga senza autenticazione, illustrata in figura 1.2, il tenente sa che il comandante è un traditore perché la sua firma appare su due differenti ordini.

Nell'algoritmo SM(m), un tenente mette la sua firma su un ordine per confermare di averlo ricevuto. Si può osservare che la firma dell'm-simo tentente non modifica in alcun modo lo stato di nessun altro tenente, per cui è superflua.

(Più precisamente è la condizione A2 che la rende inutile). In particolare nessun tenente ha bisogno di firmare l'ordine del generale in SM(1).

Proviamo ora la correttezza dell'algoritmo SM(m).

Teorema 1 *Per qualunque m, l'Algoritmo SM(m) risolve il problema dei Generali Bizantini, se il numero di traditori è al massimo m.*

Dimostrazione. Prima proviamo *IC2*. Se il comandante è leale, spedisce l'ordine *v:0* ad ogni tenente al passo (1). Ogni tenente leale riceverà perciò l'ordine al passo (2)(A). Quindi, visto che nessun tenente traditore può creare un messaggio della forma *v':0*, un tenente non riceverà ulteriori ordini al passo (2)(b). Da questo punto in poi, per ogni tenente leale i, l'insieme V_i ottenuto nel passo (2) contiene il solo ordine v, a cui ogni tenente leale obbedirà al passo (3) perché per la proprietà 1 della funzione *choice* si ha $choice(\{v\}) = v$. Questo prova *IC2*.

Perciò *IC1* segue da *IC2* se il comandante è fedele e per dimostrare *IC1* dobbiamo considerare solo il caso in cui il comandante sia un traditore. Due tenenti leali i e j obbediscono allo stesso ordine al passo (3) se gli insiemi di ordini V_i e V_j che ottengono al passo (2) sono gli stessi. Perciò, per provare *IC1* è sufficiente provare che, se i aggiunge un ordine v in V_i al passo (2), allora j deve mettere lo stesso ordine v in V_j al passo (2). Per far vedere questo, dobbiamo mostrare che j riceve un appropriato messaggio firmato contenente l'ordine. Se i riceve l'ordine v al passo (2)(A), allora lo manda a j al passo (2)(A)(ii); così j lo riceve (per la proprietà *A1* ogni messaggio spedito viene ricevuto correttamente). Se i aggiunge l'ordine V_i al passo (2)(B), allora deve ricevere un primo messaggio della forma $v : 0 : j_1 : \ldots : j_k$. Se j è uno dei j_r, allora, per la proprietà *A4* (nessuno può contraffare le firme), deve già aver ricevuto l'ordine v. Se non è così, si considerano due casi:

1. $k < m$: In questo caso, i spedisce il messaggio $v : 0 : j_1 : \ldots : j_k : i$ a j, in questo modo j riceve l'ordine v
2. $k = m$: Questo significa che il comandante è un traditore, quindi al più $m - 1$ tenenti sono traditori, quindi almeno uno dei tenenti $j_1, \ldots, j_m$ è leale. Tale tenente deve aver spedito a j il valore v quando lo ha ricevuto la prima volta, così j deve comunque aver ricevuto questo valore.

Questo conclude la prova.

Capitolo 2

Accordo bizantino con autenticazione: osservazioni e problemi da risolvere

2.1 Messaggi "orali", messaggi "scritti" e modello ibrido

Per definizione un fallimento bizantino non mette nessun tipo di limite ai possibili comportamenti di un processore.

Abbiamo però visto nel capitolo precedente che, mentre il modello dei messaggi "orali" non pone limiti a quello che un traditore può fare, il modello dei messaggi scritti pone alcune limitazioni alle possibili azioni dei generali traditori. Queste limitazioni devono essere garantite da un sistema di autenticazione.

Per comodità riportiamo le assunzioni per i messaggi orali introdotte a pag 6 (vedi [1], pag 387):

A1: Ogni messaggio spedito da due processori corretti è consegnato correttamente

A2: Il destinatario di un messaggio sa chi l'ha mandato (canali privati tra un processore e l'altro)

A3: È possibile rilevare l'assenza di un messaggio

Notiamo in particolare che la condizione *A2* vale per messaggi consegnati *direttamente*, non per messaggi che sono inoltrati da altri processori.

La proprietà *A3* è sicuramente soddisfatta se il sistema è sincrono.

Prima di dare le assunzioni per i messaggi "scritti" descriviamo brevemente un modello che tenga conto più accuratamente dei vari tipi di errore che si possono verificare in un sistema reale. Infatti nella trattazione tradizionale del problema dei fallimenti bizantini, quando si parla di numero di processori "difettosi" (o di generali traditori) non si distingue che tipo di fallimenti presentano i processori, e necessariamente in un'analisi teorica occorre considerare il caso peggiore.

Il modello "ibrido" invece classifica i processori difettosi in base alla gravità del fallimento che presentano in questo modo

processori manifest-faulty I fallimenti provocati da questi processori possono essere individuati da meccanismi presenti in ogni processore corretto (ad esempio messaggi formattati male o con errori nel calcolo della checksum o nel codice di correzione degli errori)

processori symmetric-faulty Un processore di questo tipo mostra lo stesso tipo di comportamento ad ogni processore corretto. Ad esempio può spedire un messaggio contenente un valore sbagliato, ma questo valore sarà lo stesso per tutti i processori corretti.

processori arbitrary-faulty processori di questo tipo non hanno nessun tipo di restrizione nel loro comportamento. Possono anche presentare coportamenti diversi nei confronti di differenti processori corretti.

In riferimento al modello ibrido appena descritto, le assunzioni per i "messaggi scritti" (da aggiungere alle assunzioni per i "messaggi orali") che, nella formulazione originale, avevamo già incontrato nella sezione 1.3 sono così formulate:

A4(a) I messaggi spediti da processori **non** *arbitrary-faulty* non possono essere alterati dagli altri processori

A4(b) Qualsiasi processore corretto può indentificare il processore che ha creato un messaggio, se quest'ultimo era un processo **non** *arbitrary-faulty.*

Bisogna notare che mentre la condizione *A2* riguarda i collegamenti diretti tra due processori, la assunzione *A4(b)* si riferisce al mittente originale del messaggio, che poi può essere inoltrato da altri processori.

Generalmente si assume che le assunzioni dei "messaggi scritti" possano essere garantite con un sistema crittografico a chiave pubblica, facendo sì che ogni processore "firmi" il messaggio prima di inoltrarlo. Vedremo in seguito che affinché questo sistema garantisca effettivamente che le assunzioni per i messaggi scritti siano soddisfatte sono necessari molti accorgimenti.

2.2 Definizione del problema dell'accordo bizantino in questo contesto

Il problema dell'accordo si può sempre ridurre ad uno schema in cui un mittente possiede un valore v da comunicare ad $n-1$ riceventi. Con v_p si indica il valore adottato dal processore p. Un protocollo per risolvere l'accordo in presenza di fallimenti bizantini è un algoritmo che soddisfa le seguenti condizioni:

Accordo Se i riceventi p e q sono corretti, allora concordano sul valore adottato: $v_p = v_q$.

Validità Se p è un ricevente corretto, allora il suo valore accettato v_p è

- Il valore veramente spedito, se il mittente è corretto o *symmetric-faulty.*
- Un valore convenzionale (E) se il mittente è *manifest-faulty*

2.3 Accorgimenti per l'autenticazione

2.3.1 Problemi nell'implementazione di un sistema di firme digitali

Introduciamo una notazione compatta per rappresentare i messaggi che viaggiano in un protocollo con autenticazione:
$\{\{\ldots\{v\}_p\ldots\}_q\}_r$ indica un messaggio che contiene il valore v, firmato dal processore p, ricevuto, firmato e inoltrato dai processori $\ldots, q$ e infine firmato e inoltrato dal processore r.

Le assunzioni *A4(a)* e *A4(b)*, con questa notazione sono riassunte nella seguente condizione: se il processore p è corretto, allora in nessuna fase dell'algoritmo può esistere un messaggio $\{\{\ldots\{v'\}_p\ldots\}_q\}_r$ con $v \neq v'$ se v è il messaggio originariamente spedito da p.

Si afferma generalmente che questa condizione sarebbe soddisfatta dal metodo delle firme digitali, semplicemente calcolandole e aggiungendole in coda ai messaggi. Questo sarebbe vero se un messaggio del tipo $\{\{\ldots\{v\}_p\ldots\}_q\}_r$ potesse presentarsi solo una volta durante la vita di un protocollo.

In pratica però i protocolli per raggiungere l'accordo vengono richiamati più e più volte (ad esempio ogni volta che si deve distribuire ai vari componenti la lettura di un sensore), e due esecuzioni dello stesso algoritmo possono essere in esecuzione contemporaneamente sullo stesso processore.

Può succedere quindi che un processore salvi un messaggio valido del tipo $\{\{\ldots\{v'\}_p\ldots\}_q\}_r$ e lo reimmetta durante un'esecuzione successiva dell'algoritmo, facendo sì che ogni ricevente corretto concluda che il mittente p è *arbitrary-faulty*, dal momento che osserva due messaggi provenienti da p correttamente firmati e che contengono due valori, v e v', diversi.

Perché questo problema si possa verificare non è necessario ammettere l'esistenza di attacchi intelligenti e mirati: ad esempio un processo che prende il messaggio da immettere da un buffer sbagliato quando due algoritmi di accordo sono in esecuzione simultanea, potrebbe produrre questo tipo di comportamento.

2.3.2 Proprietà necessarie di un algoritmo per l'accordo bizantino

In questa sezione presentiamo le proprietà che un sistema di autenticazione per l'accordo in presenza di fallimenti bizantini deve soddisfare.

Non commutatività

Il sistema di firme digitali non deve essere commutativo. Cioè deve valere la proprietà

$$\forall p, q, v \quad \{\{v\}_p\}_q \neq \{\{v\}_q\}_p$$

altrimenti potrebbe succedere che se p è difettoso, in un round successivo (quando il mittente è q) un processo difettoso possa immettere il messaggio $\{\{v\}_p\}_q = \{\{v\}_q\}_p$ facendo sì che q sia a torto considerato difettoso.

Verifica delle sequenze di firme

Controllare un messaggio firmato da t processori non è un compito banale. Ammettiamo che il numero di processi sia n: non è sicuramente possibile controllare tutte le possibili sequenze di t oggetti scelti tra n perché un algoritmo di questo tipo avrebbe complessità esponenziale.

Una possibilità potrebbe essere quella di aggiungere un suggerimento al messaggio ogni volta che viene firmato, aggiungendo l'identificatore del processo che inoltra il messaggio dopo averlo firmato.

Tuttavia un metodo del genere aumenta la lunghezza del messaggio di una quantità $O(n \log n)$. Questo incremento è giudicato eccessivo da Srikanth e Toueg [5] che fissano un limite superiore $O(n)$ all'allungamento del messaggio dovuto allla firma digitale.

Un altro possibile approccio prevede di stabilire un ordinamento totale sui messaggi in modo che un ricevente possa dedurre dal contesto quale sequenza di firme deve usare per la verifica.

Si assume comunque che ogni processore conosca le firme di tutti gli altri.

Distinguere sessioni concorrenti

Dal momento che diverse esecuzioni dello stesso algoritmo possono essere in esecuzione contemporaneamente, diventa importante poter determinare a quale esecuzione appartiene ciascun messaggio. Altrimenti potrebbe succedere che un processore *arbitrary-faulty* prenda un messaggio da un'esecuzione e lo usi in un'altra.

La soluzione più ovvia prevede di allegare al messaggio un identificatore della sessione (tipicamente l'identificatore del processo che l'ha iniziata). Questa soluzione incrementa la lunghezza del messaggio di $O(\log n)$ bit, e quindi è compatibile con il già citato limite fissato da Srikanth e Toueg [5].

Freshness Identifiers

Oltre a impedire che un messaggio possa passare da un'esecuzione ad un'altra concorrente ma con diverso mittente iniziale, bisogna considerare il caso di un processo bizantino che cerca di usare un messaggio di una vecchia esecuzione con lo stesso processo "iniziatore" (**replay attack**).

Per risolvere questo problema il processo che inizia una sessione dell'algoritmo di accordo deve includere in coda al messaggio un indicatore che testimoni la sua "freschezza" (**freshness identifier**). I tipi di freshness identifiers più usati sono tre e sono accuratamente descritti da Li Gong in [4].

Timestamp. Questo metodo presuppone la sincronizzazione degli orologi sui vari processi. Un tipico utilizzo del timestamp come freshness identifier è il seguente:

$$A \rightarrow B : \{Ta\}_k$$

con questa notazione si intende che A spedisce un messaggio a B accodando un timestamp Ta e firmandolo con una chiave k condivisa tra A e B. Quando B riceve il messaggio, confronta Ta con il valore corrente del suo orologio, diciamo Tb. Se $|Ta - Tb|$ è minore di un certo limite, il messaggio viene accettato.

Nei protocolli basati sui freshness identifiers si possono distinguere tre entità con compiti diversi:

supplier sceglie l'identificatore

prover Usa l'identificatore incorporandolo nel messaggio

verifier Riceve il messaggio e controlla la validità dell'identificatore

In questo caso *supplier* e *prover* sono lo stesso processo (il mittente), anche se si assume che il timestamp sia fornito dal sistema. Questo perché il formato del timestamp è di dominio pubblico, quindi il *prover* potrebbe scegliere di usare un qualunque timestamp. Il *verifier* deve fidarsi dell'onestà del *prover*, nel senso che quest'ultimo non usi intenzionalmente timestamp errati, e anche della sua competenza nel non sbagliare a scegliere il timestamp. In altre parole la sicurezza di un sistema del genere dipende dalla correttezza dei timestamp usati dal *prover*.

Dal momento che gli algoritmi per una sicura sincronizzazione degli orologi generalmente hanno bisogno di uno schema di autenticazione, ipotizzare per quest'ultimo l'esistenza di orologi sincroni è alquanto azzardato in quanto sembra portare ad un circolo vizioso. Tuttavia sembra che sia possibile risolvere questi due problemi contemporaneamente, supponendo che gli orologi hardware siano discretamente accurati [6]; inoltre esistono algoritmi diversi per la sincronizzazione degli orologi, che non richiedono questo prerequisito.

Un rischio significativo nell'usare questo metodo si presenta quando un processore difettoso può anche avere un *clock* difettoso e quindi può inviare messaggi firmati con timestamp nel futuro. Anche se poi questo processo dovesse subire una *recovery*, un altro processo difettoso potrebbe usare i vecchi messaggi quando arriva veramente il tempo segnato dai timestamp difettosi. Per far sì che messaggi di questo tipo non siano più validi dopo una recovery del processo è necessario cambiare la chiave pubblica del processo durante la recovery (cfr. *Riparare i guasti e ripartire*, pag. 15).

Numeri realmente casuali. Un altro metodo per realizzare i freshness identifiers si basa sui *numeri realmente casuali*. In un'implementazione del genere il *supplier* e il *verifier* sono lo stesso processo. Questo algoritmo richiede quindi lo scambio di due messaggi. Esistono tre tipici modi di usare numeri casuali come freshness identifier. Sia Na un numero casuale scelto da A e sia k la chiave che A e B usano per comunicare. $f()$ indica una funzione analoga all'incremento di uno.

Primo metodo

1. $A \rightarrow B : Na$
2. $B \rightarrow A : \{Na\}_k$

Secondo metodo

1. $A \rightarrow B : \{Na\}_k$
2. $B \rightarrow A : Na$

Terzo metodo

1. $A \rightarrow B : \{Na\}_k$
2. $B \rightarrow A : \{f(Na)\}_k$

Nel terzo metodo l'uso della funzione f garantisce che i messaggi 1 e 2 non siano identici. È importante che Na sia un vero numero casuale, cioè che non sia in nessun modo prevedibile (al limite neanche da A stesso) quale sarà il prossimo Na utilizzato (quindi dei numeri pseudocasuali generati da un algoritmo deterministico non vanno bene, si dovrebbe preferibilmente usare dei numeri ottenuti misurando qualche particolare fenomeno fisico).

Questi metodi garantiscono che sia estremamente difficile per un terzo processo intercettare il messaggio 1 e predire il messaggio 2.

Contatori. Invece di usare un numero casuale, in un protocollo simile a quello appena descritto si possono usare i valori crescenti di un contatore. Tuttavia dal momento che il prossimo valore del contatore è facilmente prevedibile da un terzo processo, il primo e il secondo metodo descritti non sono sicuri, infatti un processo "traditore" potrebbe ricordare il messaggio $\{Na\}_k$ ed usarlo per un *replay attack* quando osserva una seconda volta il messaggio Na. Il terzo metodo è invece sicuro, ma richiede che il *verifier* si fidi dell'onestà e della competenza del *prover*.

Come nel caso dei timestamp, un processore difettoso potrebbe usare dei valori "futuri" del contatore, quindi è necessario scegliere una nuova chiave quando si reintegra un processo nel sistema.

Riparare i guasti e ripartire

La *recovery* non è banale in un sistema esposto a fallimenti arbitrari: bisogna considerare che un processo difettoso può perdere tutte le informazioni sullo stato del sistema, compreso l'identificatore della sessione corrente, il numero di round e i freshness identifier. Le informazioni ancora presenti in un processo *arbitrary-faulty* possono essere errate, ad esempio il clock o i contatori possono essere avanti o indietro.

Non è prudente riazzerare tutti i contatori al momento di una *recovery* perché il sistema sarebbe in questo caso soggetto a *replay attacks*.

Quindi per ripristinare la sincronia tra i processori quando si reintroduce un processo nel sistema è necessario che il processo che effettua la *recovery* ottenga dagli altri processori delle informazioni firmate con *freshness identifier* che descrivono lo stato del sistema.

Un altro problema già sottolineato nella sezione dedicata ai timestamp (pag. 14) è quello di assegnare una nuova chiave a un processo che riparte, e di comunicarla a tutti gli altri processi, dal momento che si è supposto che ogni processo sia a conoscenza delle chiavi pubbliche di tutti gli altri.

Messaggi con ridondanza

Il messaggio che deve essere firmato deve contenere una sufficiente ridondanza per proteggersi contro eventuali contraffazioni. Per esempio, un processo difettoso p potrebbe scegliere un numero casuale x e spedirlo in broadcast come se fosse $\{v\}_p$. C'è una possibilita rilevante che esistano un valore v' e un processore q tali che $x = \{\{v'\}_q\}_p$; diventa così possibile per p costruire una contraffazione della firma di q, e qualsiasi altro processore che successivamente firma $\{\{v'\}_q\}_p$ viene anch'esso raggirato.

Per difendersi da questi tipi di attacchi si può aggiungere vari tipi di ridondanza ai messaggi. Ad esempio si può aggiungere una *checksum* al valore originale. Non basta invece aggiungere l'identificatore dell'esecuzione corrente perché scegliendo un valore casuale questo potrebbe essere della forma $\{id, v\}_q$ dove id è l'identificatore di una esecuzione futura, e quindi un attacco potrebbe esserci nel futuro.

2.4 Implicazioni pratiche

Ci sono due grandi classi di applicazioni che usano protocolli autenticati per l'accordo in presenza di fallimenti bizantini: sistemi sicuri che devono mantenere la coordinazione nonostante attacchi intelligenti e mirati dall'esterno (ad esempio l'architettura "Rampart" della AT&T, [7])), e sistemi integrati in applicazioni *safety-critical* (ad esempio l'architettura MAFT per il controllo del volo degli aerei, [8]).

Per quanto riguarda la prima classe di applicazioni tutte le problematiche che abbiamo illustrato sono assolutamente giustificate, perché un qualsiasi punto debole del protocollo verrà sicuramente sfruttato da un nemico intenzionato a produrre il massimo danno possibile.

Per la seconda classe di applicazioni il discorso è diverso infatti in quel caso degli attacchi intelligenti non sono considerati una possibilità seria. In questo contesto le architetture che resistono agli attacchi bizantini sono vantaggiose perché semplificano la certificazione della sicurezza del sistema. Infatti è più facile costruire un sistema che può resistere a *ogni tipo* di fallimento e mostrarne la correttezza nei limiti del massimo numero di fallimenti previsti piuttosto che prevedere tutti i possibili tipi di fallimenti che possono accadere in un particolare contesto, trovare una soluzione per ognuno di essi e dimostrare che queste soluzioni non interferiscono tra loro in modo sbagliato.

Le assunzioni dei messaggi scritti vanno contro la purezza di un tale ragionamento, infatti nella realtà i processori difettosi possono veramente fare *qualsiasi cosa* mentre assumendo che si verifichino le condizioni dei messaggi scritti si restringe in qualche modo il tipo dei possibili fallimenti. In questo caso il sistema di autenticazione deve impedire ai processi difettosi di violare le assunzioni dei messaggi scritti. Per la certificazione bisogna dimostrare con assoluta evidenza che il meccanismo di autenticazione svolge correttamente questo compito: la violazione dell'autenticazione non è solo un altro fallimento che dev'essere previsto, è la violazione delle assunzioni sotto le quali è stata provata la correttezza dell'intera architettura.

L'analisi teorica del modello dei messaggi orali e scritti porta a concludere che questi ultimi hanno il vantaggio di richiedere un numero minore di processori per resistere a un pari numero di fallimenti (i messaggi orali richiedono $3t + 1$ processori per sopportare t fallimenti, i messaggi scritti solo $t + 2$). Tuttavia entrambi i metodi hanno bisogno di $t + 1$ round nel caso peggiore, anche se i protocolli "early stopping" usano un minor numero di round quando il vero numero di fallimenti è minore di t.

In pratica però questi vantaggi hanno poca importanza perché nelle applicazioni reali viene fissato generalmente il numero di round r (tipicamente $r = 2$) e questo fa sì che in entrambi i casi il numero di fallimenti tollerati sia $r - 1$. La scelta si ripercuote però sul numero di processori necessari: i protocolli autenticati in due round richiedono tre processori per tollerare un singolo fallimento, quelli non autenticati ne richiedono quattro. Tuttavia anche questa differenza diventa poco importante se per altri scopi (ad esempio la sincronizzazione degli orologi) è richiesto un numero maggiore di processori.

2.5 Messaggi firmati con fallimenti ibridi

Il classico protocollo con autenticazione, SM(m), è quello descritto da Lamport, Pease e Shostak [1] (cfr. pag 7). Se, come in molte applicazioni, limitiamo il numero di round a 2 otteniamo un protocollo (SM(1)[1]) che ha bisogno di tre processori per tollerare un guasto. Estendiamo l'analisi di questo algoritmo al modello ibrido descritto nella sezione 2.1 che distingue tre classi di fallimenti, ottenendo un nuovo algoritmo che chiameremo SMH(1)[2].

Il protocollo SMH(r)[3] riconosce e scarta i valori *manifest-faulty*. L'autenticazione evita che processi *symmetric-faulty* possano immettere dei messaggi firmati correttamente quindi un processore *symmetric-faulty* potrà solo duplicare dei messaggi che riceve (cosa innocua) oppure introdurre messaggi non correttamente firmati, che saranno scartati. In pratica *l'autenticazione trasforma un* symmetric-fault *in un* manifest-fault.

Da queste considerazioni si deduce che solo i processori *arbitrary-faulty* devono essere contati nel calcolo della tolleranza dei guasti. Riassumendo si ha un teorema, che segue direttamente dal teorema 1 dimostrato nel primo capitolo.

Teorema 2 *Per ogni r, il protocollo SMH(r) soddisfa le proprietà di validità e accordo definite nella sez. 2.2 se $r \geq a$, dove a è il numero dei processori* arbitrary-faulty

Il problema non ha senso se non ci sono almeno due processori corretti, quindi ad $r \geq a$ va aggiunta la condizione $n > a + s + m + 1$.

Un'analisi simile fatta in [9] per il protocollo con messaggi orali in un modello ibrido, OMH(r)[4], porta alle condizioni $n \geq 2a + 2s + m + r$ e $r \geq a$. Qunidi SMH(r) e OMH(r) hanno la stessa tolleranza per quanto riguarda i round ma SMH(r) ha bisogno di meno processori per tollerare lo stesso numero di guasti. Tuttavia questa maggiore tolleranza dipende fortemente dalle assunzioni fatte per l'autenticazione e quindi da tutti gli accorgimenti descritti nella sezione 2.3.2.

[1] $SM(r)$ (da Signed Message) è il nome del protocollo descritto in [1] dove il parametro r indica il numero di round eseguiti dall'algoritmo diminuito di 1. Quindi SM(1) è il protocollo autenticato originale che esegue due round

[2] Signed Message Hybrid Protocol

[3] generalizzazione per $r+1$ round di SMH(1)

[4] Oral Message Hybrid Protocol ad $r+1$ round

Capitolo 3

Fallimenti bizantini con autenticazione nell'Atomic Broadcast

3.1 Introduzione

Adesso prenderemo in considerazione un algoritmo per il rilevamento dei fallimenti bizantini con uso di autenticazione che si appoggia sul protocollo di Atomic Broadcast sviluppato da F. Cristian, H. Agili, R. Strong e D. Dolev [3].

L'Atomic Broadcast è un protocollo dove, per un certo periodo di tempo costante Δ (chiamato tempo di terminazione del broadcast), sono rispettate le seguenti proprietà:

Atomicità Se ogni processo corretto spedisce un aggiornamento al tempo U (del proprio clock), allora quell'aggiornamento è stato generato da qualche processo ed è stato consegnato da ogni altro processo corretto sempre al tempo U.

Ordinamento Tutti gli aggiornamenti inviati da processi corretti sono spediti da ogni processo corretto nello stesso ordine.

Terminazione Ogni aggiornamento il cui broadcast è stato cominciato da un processo corretto al tempo T del proprio clock viene spedito agli altri processi corretti nel tempo $T + \Delta$ dei propri clock.

Per queste proprietà, l'Atomic Broadcast può essere utilizzato per implementare quella che viene chiamata *memoria replicata sincrona (MRS)*: una memoria distribuita e non volatile che visualizza, ad ogni ciclo di clock e ad ogni processo corretto gli stessi contenuti e che richiede un tempo Δ per completare la replicazione degli aggiornamenti. Se la MRS viene utilizzata per registrare tutte le informazione relative allo stato globale del sistema allora i processi che girano su ogni processore corretto sono a conoscenza, in ogni istante, dello stato globale del sistema che esisteva nei precedenti Δ tempi di clock. Tale memoria può essere quindi identificata come una specie di memoria condivisa con la caratteristica, però, che non rappresenta un singolo punto di fallimento per il sistema.

L'uso della MRS semplifica senza dubbio la programmazione dei processi distribuiti, in quanto libera il programmatore dal gravoso compito della copia, con accanto la relativa inconsistenza fra le informazioni locali di stato (inconsistenza che deriva dai ritardi casuali delle comunicazione nonchè dai fallimenti dei processori e dei collegamenti). Fra le assunzioni che vengono fatte vi è quella di considerare la presenza di una MRS nel sistema distribuito; i vantaggi che porta sono indubbi ed inoltre è relativamente semplice adattare i paradigmi di programmazione utilizzati per una memoria condivisa ad un sistema distribuito che fa uso della MRS.

3.2 Assunzioni

L'algoritmo si basa sulle seguenti assunzioni:

1. Si considera un sistema di processi che mantiene una MRS. I processi aggiornano il contenuto della MRS utilizzando un protocollo di atomic broadcast implementato da n processori distribuiti. Alcune coppie di processori possono comunicare fra loro attraverso collegamenti punto-a-punto. I processori che sono in collegamento diretto vengono chiamati VICINI. Viene usato il simbolo G per rappresentare la rete di comunicazione composta da n processori e da m collegamenti. Il software manager della MRS può chiedere a un processore di inviargli (utilizzando il protocollo di atomic broadcast) un aggiornamento σ tramite il comando `broadcast(`σ`)`. Per inviare sull'adiacente collegamento i un messaggio m contenente un dato aggiornamento, un processore p invoca il comando `send(m) on i`. Ogni collegamento mantiene un buffer per gestire le code dei messaggi in trasmissione e ricezione. Per ricevere il messaggio m, contenente un certo aggiornamento, proveniente dal collegamento i, un processore deve invocare il comando `receive(m) from i`. Gli aggiornamenti ricevuti vengono spediti al processo manager della MRS.

2. I processori presenti in G hanno nomi distinti ed è presente un ordinamento totale su di essi.

3. Indicheremo con F un insieme di processori e collegamenti sui quali si verifica un fallimento durante l'esecuzione del protocollo; indicheremo con G-F la restante rete contenente i processori corretti ed i collegamenti fra di essi. Nel caso che la rete G-F risultante non sia connessa, l'algoritmo non può garantire l'atomicità.

4. Ogni processore ha accesso ad un proprio clock. Indicheremo con C_p il clock del processore p e indicheremo con $C_p(t)$ il valore del clock al tempo t. Si assume che i clock dei processori corretti descrivano funzioni monotone crescenti del tempo reale e che la precisione dei clock sia abbastanza buona; in questo modo letture separate del clock produrranno valori diversi (ciò serve per garantire che nessun processore corretto produca lo stesso timestamp per due volte). Per semplicità viene imposto un limite al tempo di vita del sistema; in questo modo non si potrà verificare un

azzeramento dei clock ed i valori di quest'ultimi non rischieranno di sovrapporsi. In alternativa si può assumere che esista un ordinamento fra i timestamp che vengono generati da clock ciclici. Si assume inoltre che i clock dei processi corretti siano approssimativamente sincronizzati; cioè che per ogni coppia di processi corretti p e q e per ogni tempo reale t:

$$|C_p(t) - C_q(t)| < \epsilon$$

dove ovviamente t è lineare.

5. I processori girano sotto il controllo di un sistema operativo in real time che garantisce il multitasking. L'attivazione al tempo T di un task A con il parametro di input B avviene da parte del sistema operativo tramite il comando `schedule A(B) at T`. Tale comando non ha effetto se avviene ad un tempo $U > T$ e successive invocazioni dello stesso comando non producono effetti.

6. Per i tipi di messaggi utilizzati nel protocollo, supporremo che la trasmissione e gli eventuali ritardi dei processi siano limitati da una costante δ. Più formalmente: siano p e q due processori corretti che condividono il collegamento (corretto) i e sia r un qualsiasi altro processore corretto. Se u è il tempo reale al quale p invoca una `send(m) on i` e v è il tempo reale nel quale q finisce di ricevere il messaggio m e inizia a processarlo, allora:

$$0 < C_r(v) - C_r(u) \leq \delta$$

   Il limite δ include:

   (a) l'eventuale tempo che un messaggio m spende nella coda dei messaggi del collegamento i
   (b) il tempo necessario a trasmettere m sul collegamento i
   (c) il tempo impiegato da q per ricevere e processare m

   Nonostante il tempo aggiuntivo dovuto all'autenticazione, il tempo che impiega la CPU a processare il messaggio m è sicuramente molto più piccolo di quello che viene impiegato per trasmettere m stesso. Per semplicità non viene considerato il tempo di elaborazione della CPU. Il limite δ tiene conto inoltre dell'imprecisione dei clock nel misurare il tempo reale t; imprecisione che col passare del tempo può aumentare leggermente e che può quindi richiedere una periodica risincronizzazione dei clock stessi.

7. Per ogni componente del sistema, il numero di eventi (spedizioni, ricezioni, aggiornamenti) che si possono verificare in un certo intervallo di tempo finito, è finito. Inoltre si assume che vi sia un limite superiore al numero di aggiornamenti generati da ogni processo in una certa unità di tempo.

8. In generale un fallimento non si riferisce ad un singolo componente del sistema; se un componente è composto a sua volta da altri componenti, un fallimento di un certo tipo di uno di questi può causare un fallimento (di un altro tipo) in un altro componente. Per esempio, un clock che restituisce ad ogni richiesta lo stesso tempo, è un esempio di fallimento per *crash*; se quel clock è parte di un processore che lo utilizza per generare

timestamp associati a particolari eventi, allora il fallimento del processore può essere classificato fra quelli *bizantini.*
In questa trattazione nessun tipo di componente è considerato parte di altri. Quando si considera un determinato output, i messaggi non vengono mai suddivisi in più parti; in questo modo ogni messaggio sarà interamente corretto o interamente scorretto; possiamo così classificare senza ambiguità i fallimenti.

9. π e λ indicano rispettivamente il numero massimo di fallimenti dei processori e dei collegamenti che il protocollo potrà tollerare ad ogni passo dell'esecuzione; π e λ sono numeri interi arbitrari non negativi. Diversamente, il numero di fallimenti all'interno di uno stesso processore o collegamento non è limitato. Il tempo di terminazione Δ del protocollo è calcolato come funzione dei parametri π, λ, δ, ϵ e della massima distanza d fra due processori all'interno della rete G-F (dove F è la sottorete contenente al più π processori guasti e al più λ collegamenti guasti).

3.3 Diffusion Induction Principle

Il protocollo è basato su una tecnica di comunicazione chiamata *Information diffusion*:

1. quando un processore corretto riceve una nuova informazione, la propaga ai suoi VICINI inviando loro un messaggio.

2. se un VICINO corretto non possiede già quell'informazione, la propaga a sua volta ai propri VICINI.

In questo modo, in assenza di partizioni della rete, l'informazione si diffonde attraverso la rete giungendo a tutti i processori. C'è da sottolineare che ciò che si intende essere trasmesso con questa tecnica, non è un messaggio (per esempio una sequenza di bit), bensì una vera e propria informazione, una "**proposizione**". Per esempio un processore p può inviare al VICINO q la proposizione "*il processore s ha iniziato l'atomic broadcast di un aggiornamento σ al tempo T del suo clock*" inviando a q il messaggio `(T, s,` σ`)` contenente gli argomenti della proposizione.

I processori "**apprendono**" la verità della proposizione consegnata tramite un protocollo distribuito di ricezione dei messaggi o semplicemente monitorando lo scorrere del tempo attraverso il proprio clock.

Si consideri per esempio il caso in cui un processore p specifichi l'inizio di vari atomic broadcast nei tempi $T_1, T_2, \ldots$ ad un altro processo q. Se q riceve il broadcast T_i nel tempo $T_i + \Delta$, ma non riceve il broadcast T_{i+1} nel tempo $T_{i+1} + \Delta$, allora q apprende la proposizione "p non è corretto al tempo T_{i+1} del suo clock".

La correttezza del protocollo presentato si basa su un teorema chiamato *Diffusion Induction principle* che tratta le *proposizioni* e gli *apprendimenti* come primitive. Le proposizioni vengono considerate come oggetti primitivi di un qualche insieme fissato PROP e gli apprendimenti come una relazione primitiva sugli insiemi PROCESSORI $\times$ PROP $\times$ Real-Time, espressa informalmente come: "il processore p apprende la proposizione Φ al tempo t".

Diremo che un messaggio m trasporta la proposizione Φ al processore p se, ogni volta che p riceve m al tempo t, esiste un tempo $u \leq t$ tale che p apprende Φ al tempo u. Diremo che la proposizione Ψ si propaga (fra VICINI) se, per ogni coppia corretta di processori VICINI p e q (collegati da un collegamento corretto) e per ogni tempo u, se p apprende Ψ al tempo u, allora esiste un tempo reale v tale che q apprende Ψ al tempo v e $C_r(v) - C_r(u) \leq \delta$, dove r è un processore corretto. Per assicurare che la proposizione Ψ si propaghi, è sufficiente (ma non necessario) che p invii a q nel tempo u un messaggio che trasporta Ψ. Ciò assicura che q apprende Ψ entro δ cicli di clock dal momento in cui p apprende Ψ, a meno che q non abbia precedentemente appreso Ψ (per esempio ricevendo un altro messaggio contenente Ψ da un altro processore). Una proposizione Ψ si *diffonderà* in G se per ogni terna di processori corretti p, q, r e per ogni tempo u, se p apprende Ψ al tempo u allora esiste un tempo w al quale q apprende Ψ e $C_r(w) - C_r(u) \leq d\delta$, dove d è la distanza più grande fra due processori della rete G-F.

A questo punto possiamo enunciare e dimostrare il Diffusion Induction Principle:

> Se la proposizione Ψ si propaga fra VICINI, allora si propaga in tutta la rete G

Dimostrazione

Per ipotesi la proposizione Ψ si propaga fra vicini.
Sia r un qualsiasi processore corretto in G.
Sia d la distanza massima fra due processori di una qualsiasi sottorete di G.
Diremo che la proposizione Ψ si diffonde da p a q in una sottorete G' se, quando p apprende Ψ al tempo u, allora esiste un tempo w tale che q apprende Ψ al tempo w e

$$C_r(w) - C_r(u) \leq d(p,q)\delta$$

dove $d(p,q)$ è la distanza fra p e q nella sottorete G'.

È sufficiente provare che, *per ogni* sottorete G' composta da sole componenti corrette e *per ogni* coppia di processori p e q di G', la proposizione Ψ si diffonde da p a q. La dimostrazione verrà effettuata per induzione sul distanza $d(p,q)$.

Caso Base

Se $d(p,q) = 1$ non c'è niente da dimostrare perché p e q sono VICINI.

Passo Induttivo

Supponiamo che il teorema sia vero per $d(p,q) = (k-1)$ e dimostriamolo per k.

L'ipotesi che $k > 1$ implica che fra p e q c'è un processore corretto s tale che:

$$d(p,q) = d(p,s) + d(s,q)$$

dove le due distanze $d(p,s)$ e $d(s,q)$ sono entrambe minori di k.

Se p apprende Ψ al tempo u, per ipotesi induttiva esiste un tempo v tale che s apprende Ψ al tempo v e $C_r(v) - C_r(u) \leq d(p,s)\delta$.

Inoltre, sempre per ipotesi induttiva, esiste un tempo w tale che q apprende Ψ al tempo w e $C_r(w) - C_r(v) \leq d(s,q)\delta$..

Da cui segue quindi che $C_r(w) - C_r(u) \leq d(p,q)\delta$.

Chiameremo il tempo $d\delta$, *tempo di diffusione* della rete G-F in presenza di al più π processori guasti e λ collegamenti guasti.

3.4 Implementazione

Prima di descrivere l'implementazione pratica del protocollo elenchiamo alcune ulteriori assunzioni:

- Ogni processore p ha una funzione Φ_p di firma tale che, applicata ad una stringa x, genera una stringa y=$\Phi_p(x)$ (che verrà chiamata la firma di p su x).
- Ogni processore q conosce i nomi di tutti gli altri processori della rete e, per ogni processore $p \in$ G, q ha accesso al predicato di autenticazione $\Theta(x,p,y)$ che restituisce *true* se e solo se $y = \Phi_p(x)$.
- Se il processore q riceve la stringa (x,p,y) come parte di un messaggio m e $\Theta(x,p,y) = true$, allora è stato effettivamente il processore p a inserire la stringa (x,p,y) in m. Ovviamente se il predicato di autenticazione non funziona come dovrebbe e non rileva un messaggio contraffatto, il protocollo non è più in grado di garantire l'atomicità.

L'implementazione della funzione Φ_p e del predicato $\Theta(x,p,y)$ dipende dal tipo di corruzione si pensa possa avvenire nei messaggi.

Se la corruzione non è intenzionale (per esempio errori di trasmissione dovuti a rumori di fondo o a malfunzionamenti del collegamento) basta un semplice algoritmo per la rilevazione e la correzione degli errori.

Se la corruzione è intenzionale (per esempio un atto di sabotaggio), allora occorrono schemi più complessi e sicuri quali per esempio gli algoritmi di cifratura a chiave pubblica.

Ovviamente nessuno schema è sicuro al 100% e quindi sussiste sempre una lieve possibilità che un messaggio contraffatto venga considerato corretto.

L'implementazione dell'autenticazione dei messaggi viene trattata utilizzando tre procedure: "sign", "cosign" e "authenticate". Inoltre viene definita una struttura dati apposita per i messaggi:

```
type Smsg =
record case: tag: (first, relayed) of
    first: (timestamp: Time; update: Update);
    relayed: (incoming:Smsg);
    orocid:Processor;
    signature: string;
end;
```

Un messaggio di tipo Smsg che è stato "firmato" da k processori $p_1, \ldots, p_k$ avrà la struttura: (relayed, ... (relayed,(first, T, σ, p_1, s_1), p_2, s_2), ... p_k, s_k) dove T e σ sono rispettivamente il timestamp e l'aggiornamento inseriti dal processore p_1; s_i sono le firme1.

Una procedura `Sign` viene invocata dal creatore del broadcast (T, s, σ) e restituisce un messaggio $x = (first, T, s, \sigma, myid, fmyid(x.tag, T, \sigma))$ dove $myid$ è l'identificatore del processore e $fmyid$ è la funzione che genera la firma.

La procedura `Co-sign` viene invocata dal processore r per ritrasmettere il messaggio x già firmato da altri processori; questa procedura restituisce un nuovo messaggio y che contiene in fondo alla lista delle firme la firma di r.

La procedura `Authenticate` verifica l'autenticità di un messaggio ricevuto. Essa restituisce `false` se è avvenuta un'alterazione al messaggio originale. Se non viene rinvenuta nessuna alterazione viene restituito il valore `true` e ai restanti parametri di output T, σ, S vengono assegnati rispettivamente il timestamp, l'aggiornamento originale incluso nel messaggio e la sequenza dei nome dei processori che hanno firmato il messaggio. L'identità del processore creatore è il primo elemento della sequenza mentre il numero di salti (collegamenti) attraversati dal messaggio è la lunghezza della sequenza.

Su ciascun processore sono sempre in esecuzione concorrente tre Task:

Start Task si occupa di far partire l'atomic broadcast

Relay task che si occupa di inoltrare i messaggi ai VICINI attraverso l'atomic broadcast

Delivery Task che si occupa di spedire gli aggiornamenti del broadcast alla MRS.

Tutti i Task di un processo hanno accesso a una costante L contenente l'elenco di tutti i collegamenti di un processore.

Tutti i Task di tutti i processori hanno accesso alla costante Δ (tempo di terminazione del broadcast)

Task Start

Un processo fa partire il broadcast di un dato aggiornamento σ invocando il comando `broadcast(`σ`)` sul suo processore. Questo comando attiva il Task Start. Il broadcast di σ viene identificato dal tempo locale T nel quale è stato ricevuto dal task. Si richiama quindi la procedura `Sign` per applicare la firma; l'output risultante viene inviato su tutti i collegamenti di uscita elencati in L.

La funzione di firma non deve essere considerata un'istruzione atomica; può succedere che un processore guasto impedisca che un messaggio venga spedito su tutti i collegamenti. A questo punto il fatto che il broadcast di σ sia stato avviato viene registrato nella variabile H (variabile `history`), un insieme composto da tre elementi:

var H: Time $\times$ Processor $\times$ (Update $\cup \{\perp\}$)

dove il simbolo $\perp$ denota l'aggiornamento nullo (*null*) e servirà per rilevare un fallimento nel mittente.

Una volta che la variabile H è stata aggiornata, il Task Delivery viene messo in coda per l'esecuzione (scheduling), passandogli come parametro il tempo reale T.

Task Relay

Il Task Relay lavora come segue. Non appena viene ricevuto un messaggio, questo viene controllato tramite la procedure `Authenticate` e, nel caso sia corrotto, viene scartato.

Viene esaminata la sequenza delle firme dei processori che hanno accettato il messaggio per controllare che non ci siano duplicati; se c'è qualche firma duplicata, il messaggio viene scartato. Non appena le firme dei processori sono state autenticate, il numero delle firme $|S|$ può essere utilizzato come contatore dei salti effettuati dal messaggio attraverso la rete al fine di determinare la "tempestività" del messaggio stesso.

Se il messaggio arrivato è autentico, non ci sono firme duplicate, ed è arrivato nel tempo giusto, allora viene esaminata la variabile H che contiene la storia per determinare se il messaggio è il primo di un nuovo broadcast.

In questo caso la variabile H viene aggiornata con le informazioni che il processore mittente $s = first(S)$ ha inviato nell'aggiornamento σ al tempo T; a questo punto il Task Delivery è pronto per processare e possibilmente inviare l'aggiornamento ricevuto, entro il tempo $T + \Delta$; al messaggio ricevuto viene applicata la procedura `Co-sign` e poi viene spedito.

Altrimenti se l'aggiornamento ricevuto σ è già stato registrato in H, viene semplicemente scartato. Quindi, nel caso in cui σ sia il secondo aggiornamento per il broadcast (T, s), *deve essere avvenuto un fallimento nel mittente.* Questo avvenimento viene registrato settando $H = (T, s, \perp)$. Viene poi applicata la procedura `Co-sign` e il messaggio viene spedito in modo che gli altri processori corretti possano apprendere del fallimento del mittente.

Alla fine, se σ è associato ad un broadcast del quale si conosce il fallimento del mittente (associato cioè ad un valore `null` dell'aggiornamento), viene semplicemente scartato (13).

Task Delivery

Il Deliver Task spedisce al tempo locale $T + \Delta$ tutti gli aggiornamenti inviati correttamente al tempo T. Se esattamente un aggiornamento è stato accettato per un broadcast iniziato al tempo T, allora viene spedito. In entrambi i casi l'aggiornamento viene cancellato da H.

3.5 Correttezza del protocollo

Il protocollo realizzato diffonde due tipi di proposizioni:

1. una proposizione $\Psi(T, s, \sigma)$ della forma "*il processore s ha effettuato il broadcast dell'aggiornamento σ al tempo T*"

2. una proposizione $\Phi(T, s, \sigma)$ della forma "*O $\Psi(T, s, \sigma)$ oppure ci sono due distinti aggiornamenti σ_1 e σ_2 tali che il processore s ha avviato il broadcast di questi aggiornamenti* con lo stesso *timestamp T*"

Un processore corretto *apprende* $\Psi(T, s.\sigma)$ al tempo t se

1. $(p = s)$ $\wedge$ p inserisce(T, s, σ) nella propria storia al tempo t, oppure

2. $(p \neq s)$ $\wedge$ p riceve un messaggio x tale che la procedura di autenticazione termina con successo restituendo $(true, T, \sigma, S)$, s=first(S), e la processazione di x risulta in un aggiornamento memorizzato nella storia locale (in pratica aggiungendo (T, s, σ) o $(T, s, \perp)$ a H).

Per ogni aggiornamento σ, un processore corretto p apprende $\Phi(T, s, \sigma)$ al tempo t se ha appreso $\Psi(T, s, \sigma)$ al tempo t oppure esistono due distinti aggiornamenti σ_1 e σ_2 tali che p *apprende* $\Psi(T, s, \sigma_1)$ ad un tempo $u \leq t$ ed *apprende* $\Psi(T, s, \sigma_2)$ al tempo t. Così se $(p = s) \wedge p$ è corretto, non è possibile per p apprendere che ha inviato due differenti aggiornamenti con un identico timestamp.

Il processore corretto p *apprende* $\Phi(T, s, \sigma)$ se, e solo se:

1. p aggiunge (T, s, σ) ad H al tempo t
2. p riceve al tempo t un messaggio x tale che la procedura di autenticazione si è conclusa con successo restituendo $(true, T, \sigma, S)$, s=first(S), e la processazione di x risulta nell'azione $H(T)(s) :=\perp$.

Si può quindi dimostrare il seguente

Teorema 3 *Il protocollo sopra esposto possiede le proprietà di terminazione, atomicità e ordinamento.*

Bibliografia

[1] Leslie Lamport, Robert Shostak and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, July 1982

[2] Li Gong, Patrick Lincoln and John Rushby. Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults, 1995

[3] Flaviu Cristian, Houtan Aghili, Ray Strong, Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *IBM Alamaden Research Center*, March 1994

[4] Li Gong. Variations on the themes of message freshness and replay. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 131-136, Franconia, NH, June 1993. IEEE Computer Society.

[5] T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80-94, 1987.

[6] M. Reiter, K. Birman, and R. van Renesse. Fault-tolerant Key distribution. *TR 93-1325, Departmemt of Computer Science, Cornell University, Ithaca*, New York. January 1993.

[7] Michael Reiter. A secure group membership protocol. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 176-189, Oakland, CA, May 1994. IEEE Computer Society.

[8] R.M. Kieckhafer, C. J. Walter, A.M. Finn, and P.M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398-405, April 1988.

[9] Patrick Lincoln and John Rusby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23* , pages 402-411.