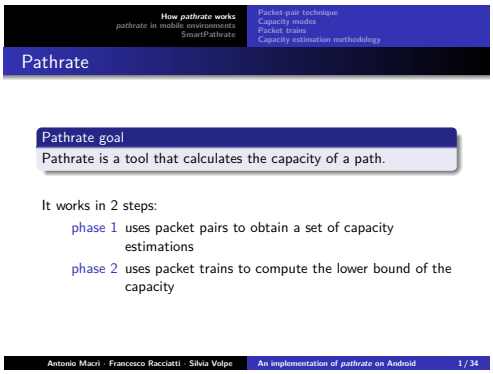
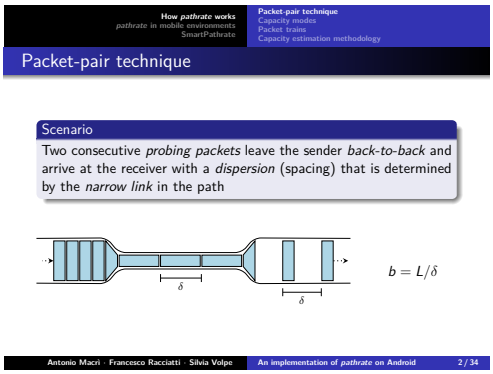
Section 1 How does pathrate work

Il tool da noi esaminato, Pathrate, misura la capacità di un path operando fondamentalmente in 2 fasi: la prima fase consiste nell'inviare coppie di pacchetti da cui determinare un insieme di stime di capacità per il path esaminato, mentre nella seconda vengono inviati treni di pacchetti per stabilire un lower bound per la capacità calcolata.



1.1/1(4)

La tecnica del *packet-pair* utilizzata da Pathrate consiste nel trasmettere consecutivamente una coppia di pacchetti in modalità back-to-back. L'assunzione sulla quale si fonda il lavoro è che i pacchetti seguano lo stesso percorso. All'interno di tale percorso il link dotato di minor capacità (transmission rate) prende il nome di *narrow link*.

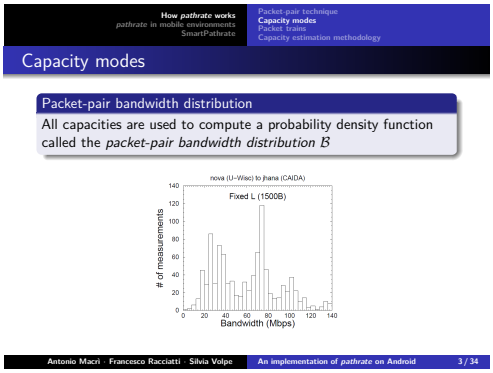


1.2/2(5)

I pacchetti della coppia arriveranno al ricevitore con una certa dispersione causata dalla presenza del narrow link seguito da uno o più link con maggiore capacità. La dispersione δ è l'intervallo temporale tra la ricezione dell'ultimo bit del primo pacchetto e dell'ultimo bit del secondo pacchetto. Dato che la dimensione dei pacchetti è nota, dalla misura della dispersione (ricavata etichettando ogni pacchetto con il timestamp d'arrivo) è possibile ricavare la capacità del narrow link e quindi dell'intero percorso:

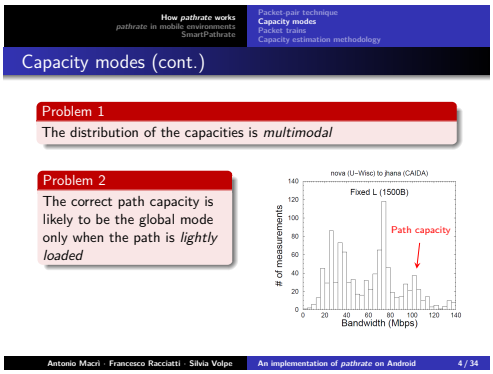
$$b = \frac{L}{\delta}$$

Le misure delle capacità tra le coppie di pacchetti generano una funzione di distribuzione di probabilità detta *packet pair bandwitdh distribution*. Nel caso ideale la distribuzione presenta un'unica moda che è l'effettiva capacità del canale.



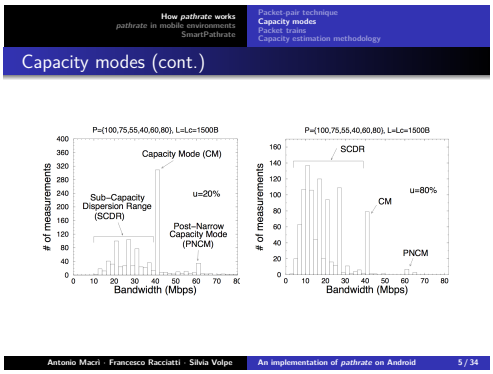
1.3/3(6)

Nel caso reale si riscontrano diversi problemi. Innanzitutto la distribuzione delle capacità risulta multimodale. In tale distribuzione la moda globale rappresenta comunque la reale capacità del canale solo nel caso in cui la rete sia poco carica. In una misurazione reale infatti vediamo come la capacità effettiva del path non sia la moda globale (70-80 Mbps) ma sia una moda locale (100Mbps).



1.4/4(7)

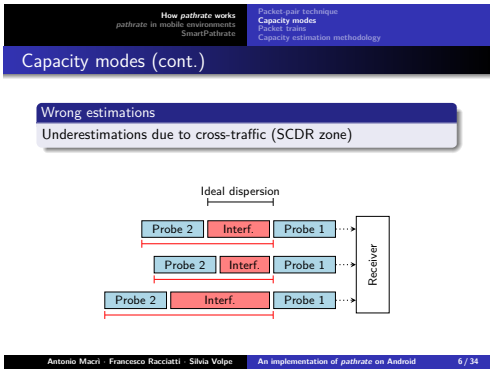
In quest'altro esempio si mostra come nella rete poco carica (livello di traffico al 20%) la moda globale indichi l'effettiva capacità del canale e come questa si distingua nettamente della altre mode locali. Sulla stesso canale, in condizioni di traffico elevato (path carico all'80 %), si nota che la capacità del canale è una moda locale ma non quella globale. Le altre modeche si formanoi si concentrano in 2 zone: la zona detta *Sub Capacity Dispersion Range* (SCDR) che comprende tutte le mode che costituiscono una sottostima della capacità reale e la zona in cui le mode locali costituiscono una sovrastima della capacità effettiva, le quali sono dette *Post Narrow Capacity Modes* (PNCM).



1.5/5(8)

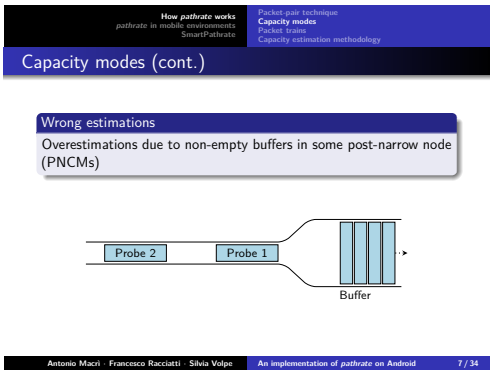
Le stime errate della capacità sono quindi dovute alla presenza di traffico di altre applicazioni.

Le mode SCDR sono dovute a cross-traffic che si insinua tra i pacchetti della coppia causando un aumento della dispersione tra i due e, conseguentemente, una sottostima della reale capacità del canale.



1.6/6(9)

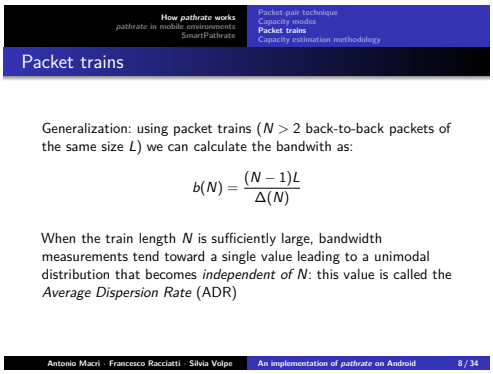
Le mode PNCM si formano invece quando il traffico di pacchetti esterni alla coppia riempie un buffer di un router di un link post-narrow, in quanto i pacchetti accumulati nel buffer determinano una ricompressione (un riavvicinamento) dei pacchetti della coppia dopo il narrow link andando così ad annullare la dispersione accumulata e causando quindi la sovrastima della capacità.



1.7/7(10)

Possiamo generalizzare la tecnica del packet pair inviando un numero N>2 di pacchetti aventi la stessa dimensione L e misurando la dispersione totale dall'ultimo bit del primo pacchetto all'ultimo bit dell'ultimo pacchetto. In assenza di cross traffic la banda viene calcolata secondo la formula:

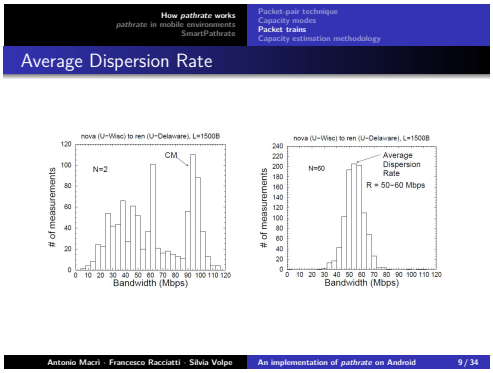
$$b(N) = \frac{(N-1)L}{\Delta(N)}$$



1.8/8(11)

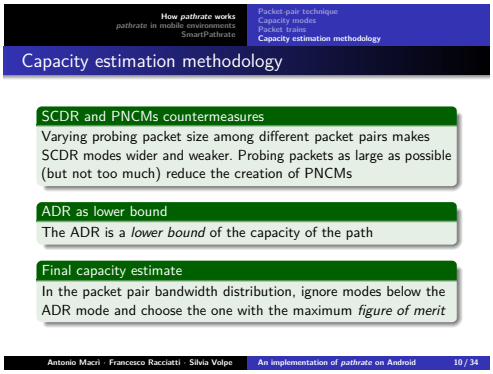
Nel caso reale, all'aumentare di N aumenta la probabilità che il cross-traffic interferisca con i pacchetti del treno portando a una sottostima della reale capacità. Particolarità dell'utilizzo dei treni è che, con N sufficientemente grande le misure delle capacità tendono a un singolo valore che rende la distribuzione *unimodale* e *indipendente da N*. Tale moda risultante è detta *Average Dispersion Rate* (ADR) e costituisce un limite inferiore per la capacità.

Nell'esempio vediamo come nel caso di una coppia di pacchetti la distribuzione sia multimodale e come la moda globale indichi la capacità effettiva del path (100Mbps), mentre invece nel caso di un treno di pacchetti (N=60) la distribuzione diventi unimodale. Il valore di tale moda costituisce una sottostima della capacità (50-60Mbps).



1.9/9(12)

In Pathrate vengono utilizzate diverse strategie per individuare correttamente la capacità del path: innanzitutto si cerca di rendere più deboli le SCDR variando la dimensione dei pacchetti fra coppie diverse, mentre si limita la formazione di mode PMCM utilizzando dei pacchetti di grandi dimensioni. La dimensione dei pacchetti non deve essere però eccessivamente grande, in modo da non favorire la formazione delle SCDR dovuta all'aumentare della probabilità di cross traffic. In secondo luogo si utilizza l'ADR come lower bound della capacità e infine si va a calcolare la capacità scartando tutte le mode con valore inferiore a quello dell'ADR e scegliendo quella con maggiore cifra di merito dopo l'ADR.

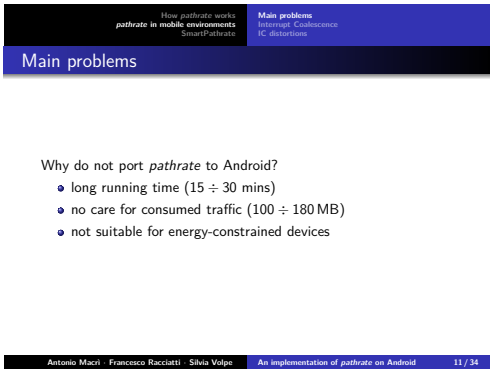


1.10/10(13)

Section 2 Why pathrate doesn't work in mobile environments

Pathrate non è direttamente portabile in ambiente mobile

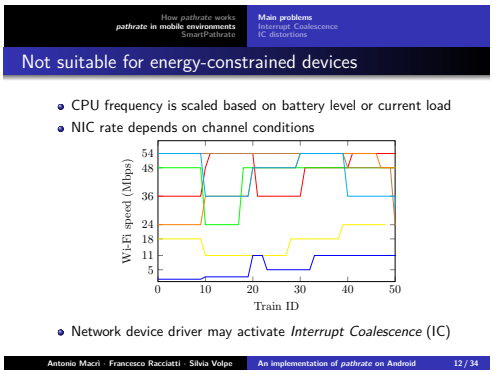
- innanzitutto per l'elevato tempo di risposta dell'applicazione, che caria tra i 15 e i 30 minuti;
- in secondo luogo perchè può arrivare a trasmettere fino a 180MB, non prendendo in considerazione eventuali limiti di traffico imposti dal piano tariffario dell'utente;
- non è adatto per dispositivi con vincoli energetici.



2.1/11(15)

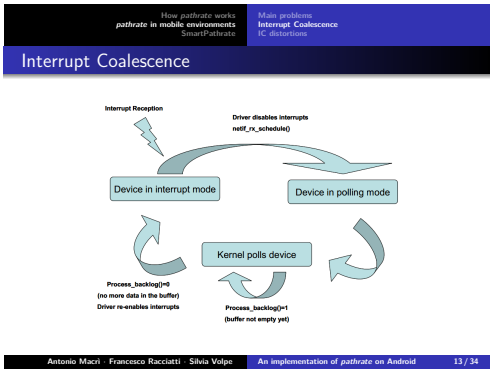
Pathrate non è progettato per funzionare su dispositivi con vincoli energetici. Infatti non prende in considerazione le seguenti caratteristiche:

- la frequenza della CPU dipende dal livello della batteria o dal carico della CPU stessa
- la velocità della NIC si adatta alle condizioni del canale (in figura sono mostrate misurazioni della velocità della NIC durante diverse esecuzioni di SmartPathrate. La capacità del canale viene calcolata ad ogni round e si nota l'estrema variabilità della capacità del canale WiFi all'interno di un'unica esecuzione.);
- i driver dei dispositivi di rete possono attivare il meccanismo di *Interrupt Coalescence*



2.2/12(16)

2.3/13(17)



L'Interrupt Coalescence è un meccanismo utilizzato sui ricevitori NAPI-compliant per evitare che il sistema operativo venga inondato da un'eccessiva quantità di interruzioni sollevate dalla NIC. L'IC si basa sul meccanismo di *polling*.

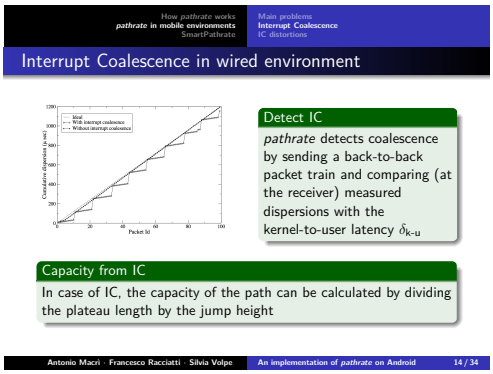
All'arrivo del primo pacchetto la NIC lo memorizza in un buffer in RAM via DMA, solleva la prima interruzione che viene catturata dal gestore delle interruzioni. Nel classico meccanismo ad interruzione il driver fa in modo che il pacchetta raggiunga l'applicazione. Nel meccanismo di IC il driver maschera le interruzioni e mette l'interfaccia il *pollin mode* inserendo il suo descrittore in una lista detta *polling list*. A intervalli regolari il kernel esaminerà la lista di polling e, a seconda del meccanismo di scheduling utilizzato servirà un'interfaccia. Preleverà quindi i pacchetti memorizzati nel relativo buffer e li consegnerà all'applicazione.

Quindi, a differenza del meccanismo ad interruzione, con l'IC l'applicazione riceve un insieme di pacchetti e non uno alla volta.

Se il buffer non si svuota l'interfaccia resterà in stato di polling. Se il buffer invece si svuota completamente allora il driver toglie l'intefaccia dalla polling list e riabilita le interruzioni.

L'utilizzo di un meccanismo di questo tipo consente di ridurre significativamente l'overhead causato dai continui cambi di contesto che andrebbero a prodursi se la scheda di rete sollevasse le interruzioni all'arrivo di ogni pacchetto.

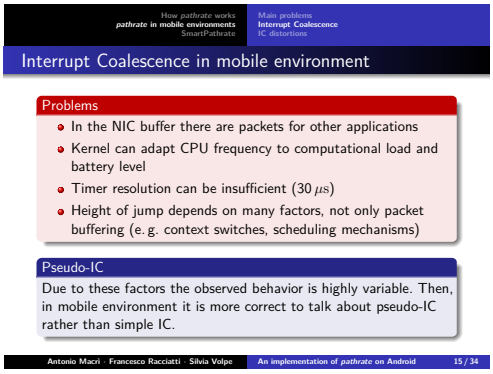
2.4/14(18)



Come visibile, su una GigabitEthernet il pattern delle dispersioni tra pacchetti consecutivi è estremamente regolare. Pathrate rileva la coalescenza inviando un treno di pacchetti back to back e comparando sul ricevitore le misure delle dispersione con la latenza kernel to user δ_{k-u}. Le coppie di pacchetti che presentano dispersione comparabile con la latenza kernel to user sono quelle che sperimentano coalescenza In ambiente wired, in caso di IC la capacità del link può essere calcolata dividendo la lunghezza del plateau per l'altezza dei salti.

Si delineano però diversi problemi, soprattutto in ambiente mobile:

- il buffer della NIC contiene pacchetti di altre applicazioni, e più in generale l'ambiente wired è più facilmente controllabile rispetto all'ambiente mobile.
- il kernel può adattare la frequenza della CPU al carico computazionale e al livello di carica residua della batteria
- la risoluzione del timer può essere insufficiente. Da prove effettuate la risoluzione del timer dei disositivi mobili si attesta sui 30 μs
- diminuire la risoluzione (aumentando la frequenza di aggiornamento del timer) significa aumentare sensibilmente il consumo energetico
- l'altezza dei salti dipende da un insieme di fattori e non solo dall'intervallo di polling (per esempio i salti generati dai cambi di contesto o dai meccanismi di scheduling che operano sulla polling list)

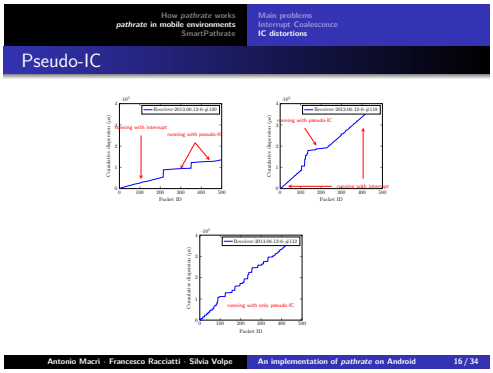


$2.5/15_{(19)}$

la risoluzione di un timer è l'intervallo minimo di tempo che riesce a misurare

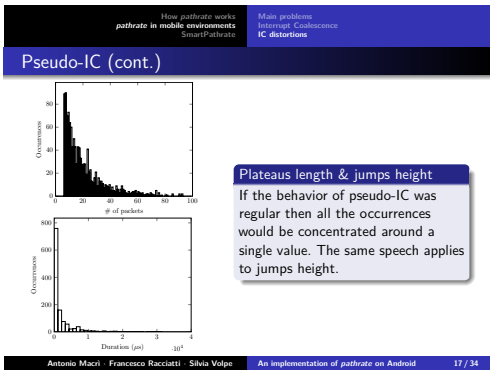
Data la variabilità del comportamento dell'applicazione osservato, in ambiente mobile è più corretto parlare di pseudo-IC piuttosto che di IC.

Sono mostrate le misurazioni delle dispersioni ottenute da prove reali. Si osserva l'estrema variabilità del comportamento.



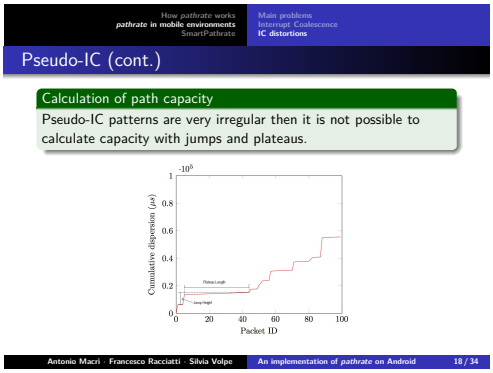
2.6/16(20)

In questa slide sono mostrate le distribuzioni sulla lunghezza dei plateau (in alto) e sulla durata dei salti (in basso). Si vede che oltre l'80% dei plateau ha una lunghezza inferiore a 40 pacchetti. Risulta evidente inoltre che la distribuzione non si concentra attorno a un valore, come ci si aspetterebbe in caso di coalescenza.



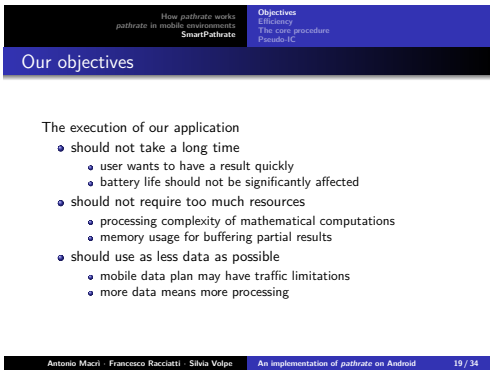
2.7/17(21)

In ambiente mobile il pattern di pseudo-IC è molto irregolare per cui non è possibile calcolare le capacità usando i salti e i plateau.



2.8/18(22)

Section 3 SmartPathrate

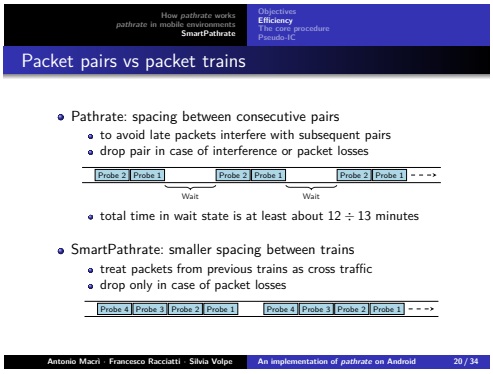


3.1/19(24)

Dato che viene eseguita su dispositivi mobili, un requisito fondamentale dell'applicazione è la rapidità di risposta, sia per questioni di usabilità (l'utente non vuole attendere troppo), sia perché, se incorporata in altri applicativi (per esempio per lo streaming video, cioè non usata come applicazione *standalone*) può essere *richiesta* una certa prontezza.

Un altro aspetto è l'impiego di risorse, in termini di CPU e RAM. Nell'elaborazione dei dati vengono usati diversi algoritmi matematici, alcuni abbastanza complessi. Bisogna poi memorizzare i dati (parziali). Girando su dispositivi in cui entrambe le risorse sono piuttosto contenute, bisogna cercare di limitarne l'uso il più possibile.

Anche la "data complexity" è bene che sia ridotta al minimo. Il piano tariffario in uso può prevedere un limite sul traffico e, per di più, all'aumentare della dimensione dei dati raccolti aumenta naturalmente il tempo impiegato ad elaborarli (e non è detto che cresca in maniera lineare).



3.2/20(25)

Vediamo quindi come possono essere affrontati e risolti questi problemi. Innanzitutto il tempo di esecuzione.

Un problema basilare di `pathrate` è l'uso delle *coppie di pacchetti*, non come tecnica, ma come "implementazione", cioè inviare letteralmente coppie di pacchetti. Ciò vuol dire inviare due pacchetti in sequenza e poi attendere un po', prima di inviarne un'altra, in modo da smaltire eventuali pacchetti ritardatari che altrimenti inficerebbero la ricezione dei treni seguenti, i quali verrebbero riconosciuti come "bad train" e scartati.[1]

Il problema è che `pathrate` aspetta almeno mezzo secondo (in realtà di $1.25 \times RTT$): siccome in tutto si inviano come minimo 1500 tra coppie e treni di pacchetti,[2] ci sono $12 \div 13$ minuti di attesa in semplice stato di *sleep*.

Questo è uno dei motivi per cui inviamo treni di pacchetti, riducendo inoltre il tempo di attesa tra un treno e l'altro.[3] In più, decretiamo di avere un "bad

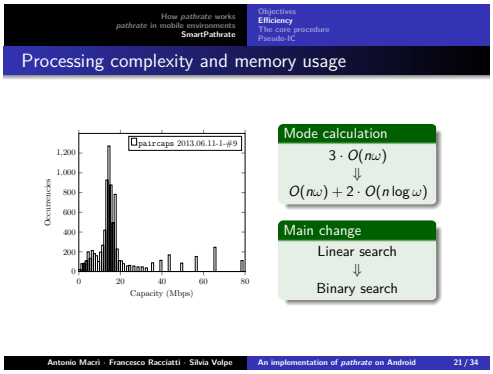
[1] `Pathrate`, infatti, scarta il treno sia quando si perdono pacchetti sia quando arrivano pacchetti di un vecchio treno. Hanno un atteggiamento "conservativo", ossia cercano di avere una situazione il più possibile pulita, prima di fare misure.

[2] In questo contesto, le coppie di pacchetti sono semplicemente considerate come treni di lunghezza 2.

[3] Il quale è comunque necessario altrimenti i pacchetti potrebbero subire dropping.

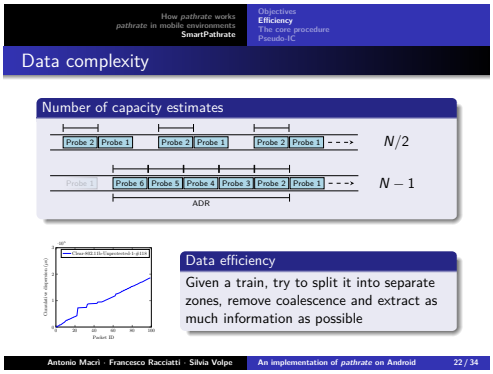
train" solo quando si perde qualche pacchetto: eventuali pacchetti di vecchi treni li consideriamo alla stregua del cross traffic.[4]

3.3/21(26)



Le elaborazioni sui dati raccolti non avvengono istantaneamente, ma richiedono un po' di calcoli. Per esempio, l'algoritmo per trovare le mode ha un impatto non trascurabile affatto sui tempi. Se indichiamo con n il numero di misure di capacità e con ω il bin width, il loro algoritmo originario aveva una complessità dell'ordine di $3 \cdot O(n\omega)$. Modificando alcune porzioni dell'algoritmo siamo riusciti a ridurre la complessità a qualcosa come $O(n\omega) + 2 \cdot O(n \log \omega)$. In particolare, siamo intervenuti sul calcolo della campana associata a una moda, trasformando due ricerche lineari in binarie. Non si tratta di un netto abbattimento ma certamente è un miglioramento, che assieme ad altri piccole ottimizzazioni hanno consentito di tagliare di molto i tempi.[5]

3.4/22(27)

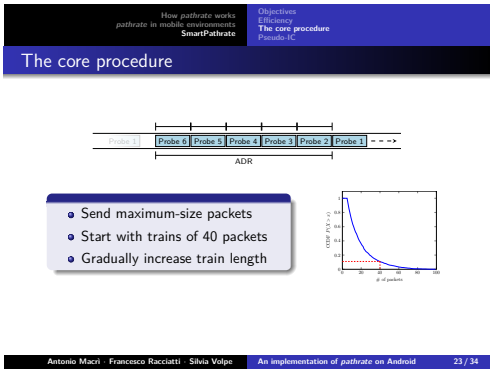


L'uso di coppie di pacchetti, anziché di treni, è in un certo senso meno *efficiente in termini di dati*. Per ogni coppia di pacchetti si ottiene una sola stima di capacità. Spedendo un treno di N pacchetti si ottengono $N - 1$ stime di capacità (contro le $N/2$ che si sarebbero ottenute inviando lo stesso numero di pacchetti in coppie). Il nostro obiettivo è riuscire ad estrarre il maggior numero di dati dal minore numero di treni. Per questo motivo usiamo gli stessi treni anche per stimare l'ADR. È particolarmente importante per noi riuscire a trattare i treni pur nella loro complessità e variabilità, scartare il minor numeri di treni, e lavorare al meglio su quelli ricevuti, separando le zone di coalescenza da quelle "utili", dalle quali si possono ricavare stime di capacità.

[4] Se siamo fortunati, vecchi pacchetti li riceviamo appena prima di ricevere il treno successivo, quindi non influenzano le misure sulle dispersioni dei pacchetti seguenti.

[5] Si può dire che il codice è l'evidente risultato della stratificazione di più interventi, compiuti da più mani: dall'originale `pathload` a `pathrate`.

Cominciamo a esporre il funzionamento a grandi linee dell'applicazione. Innanzitutto, inviamo treni della massima dimensione (per i motivi già spiegati di ridurre gli errori di misura, risoluzione del timer, eccetera). Partiamo con treni di dimensione minima (perché all'inizio si conosce la qualità del canale e treni troppo lunghi potrebbero non essere ricevuti) e via via la aumentiamo, fino a raggiungere il massimo.

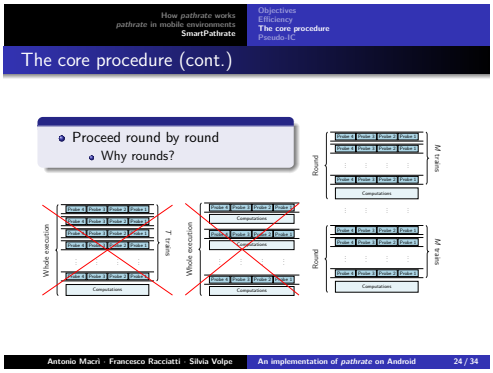


3.5/23(28)

Il valore minimo è di 40 pacchetti perché, come già accennato qualche slide fa, oltre l'80 percento dei plateau hanno dimensione inferiore a 40 pacchetti, e questo ci consente di evitare, nella maggior parte dei casi, di cadere completamente in una zona di coalescenza. La dimensione viene poi incrementata gradualmente, sia perché treni più lunghi significano maggiore probabilità di evitare la coalescenza, sia perché si può incentivare l'interfaccia ad accelerare, sia perché si ottiene una stima migliore dell'ADR.[6]

Una caratteristica della nostra applicazione è che l'esecuzione procede per *round*.

Lo schema usato in `pathrate` prevede di inviare un numero prefissato di coppie di pacchetti e di treni e poi eseguire i calcoli solo alla fine. Il problema di questo approccio è che noi vogliamo far sì che l'applicazione si interrompa appena riesce a dare una stima accettabile. Chiaramente serve un *feedback*, che si può ricavare dall'analisi dei risultati parziali.



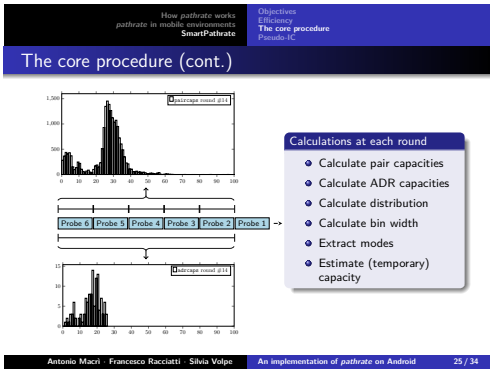
3.6/24(29)

Si potrebbe allora pensare all'approccio opposto: ogni volta che si invia un treno si fanno anche i calcoli. Tuttavia, in questo modo si finisce per fare troppi calcoli, pochi dei quali effettivamente utili (è difficile che un singolo treno sia decisivo).

La soluzione migliore è allora inviare prima un certo numero di treni e poi ricavare i risultati parziali: questo è un round. Peraltro, aspettiamo di raccogliere almeno un numero minimo di capacità (1000), prima di iniziare a dare una stima di capacità.

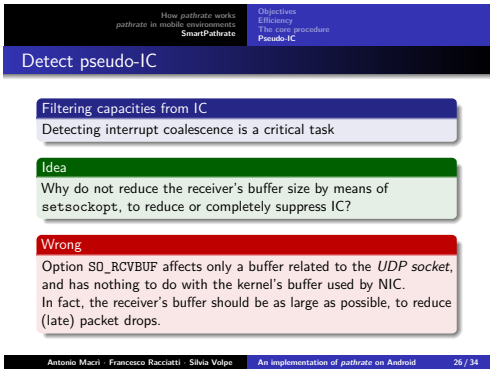
[6] È anche vero che il kernel potrebbe rispondere con la coalescenza.

In ciascun round si elaborano tutti i treni ricevuti in quel round. Per ogni treno si estraggono, da un lato, le dispersioni tra pacchetti e le relative capacità (*pair capacities*); dall'altro, la dispersione dell'intero treno e la relativa capacità (*ADR capacity*), che ci servirà a ricavare la stima dell'ADR. Quindi si mettono insieme tutte queste capacità in due distribuzioni distinte, si calcolano le ampiezze dei rispettivi bin, si estraggono le mode e si dà infine una stima (temporanea) della capacità del path.



3.7/25(30)

Si capisce che, affinché l'algoritmo funzioni correttamente, è di cruciale importanza riuscire a rilevare la coalescenza al meglio. A questo punto potrebbe sorgere una domanda: se la coalescenza è dovuta al fatto che i pacchetti vengono accumulati nel buffer, perché non si prova a ridurre la dimensione del buffer, usando la `setsockopt()`? Con un buffer capace di contenere un solo pacchetto, si potrebbe pensare di annullare completamente la coalescenza e forzare il kernel a consegnare immediatamente i pacchetti al processo.

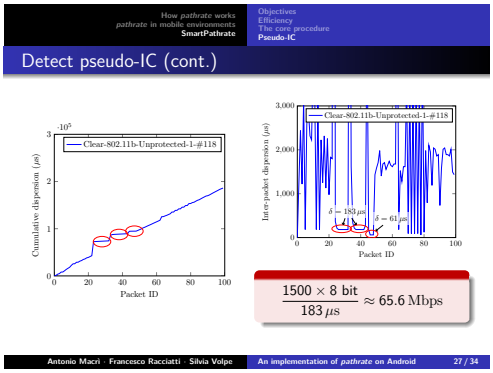


3.8/26(31)

In realtà, impostando il `SO_RCVBUF` viene solamente assegnato il valore di un contatore associato al socket UDP, quindi qualcosa che è a livello di trasporto (quasi applicazione), che determina solo la quantità massima di dati che l'applicazione può prelevare in una volta: se viene superata, l'azione intrapresa dal kernel è scartare i pacchetti (ormai ricevuti e che hanno attraversato tutto lo stack TCP/IP).[7]

[7] La dimensione del buffer è un'informazione associata al socket UDP/TCP, che non potrebbe esser nota ai livelli inferiori.

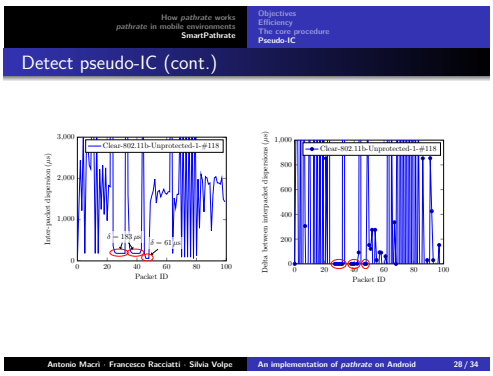
3.9/27(32)



Usare treni è peraltro una scelta praticamente obbligata, perché usando coppie di pacchetti verrebbe difficile rilevare correttamente la coalescenza. Il codice di `pathrate` prevede di rilevare la coalescenza *tout court*, ovvero: inviato un treno, controllano se almeno il 60% delle dispersioni tra pacchetti successivi ricadono entro 2.5 volte la $\delta_{\text{k-u}}$. Se così avviene, allora decretano che c'è coalescenza e calcolano la capacità dividendo i dati inviati nel plateau per il salto. Noi dobbiamo invece suddividere il treno rimuovendo i tratti di coalescenza.

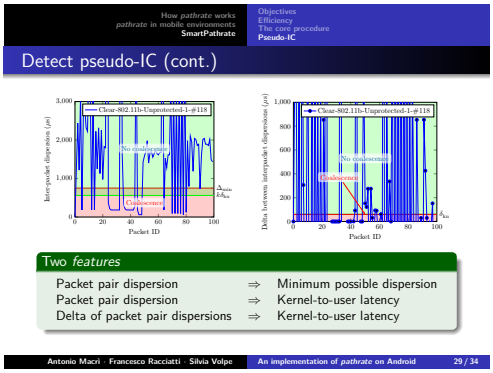
Nei nostri esperimenti, abbiamo visto che non basta considerare la latenza kernel-utente. Il motivo fondamentale è la velocità della CPU che varia, per cui in momenti diversi la $\delta_{\text{k-u}}$ assume valori diversi. In questo grafico sono distinguibili a occhio tre tratti di coalescenza. Calcolando le dispersioni tra pacchetti successivi, otteniamo il grafico riportato a destra in alto. Nei primi due tratti, la latenza è circa $180\,\mu$s, mentre nel terzo vale appena $60\,\mu$s (e in altre misurazioni la differenza è altrettanto marcata). Il problema è che la velocità ridotta del kernel può confondersi con una alta velocità della rete: $180\,\mu$s corrispondono al tempo di ricezione di un pacchetto di 1500 byte con una velocità di 66 Mbps, pienamente raggiungibile con una rete 802.11n.

3.10/28(33)



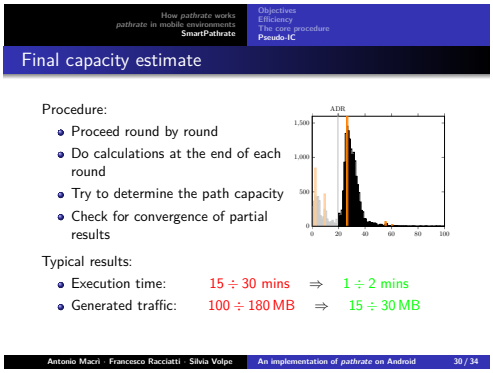
Si può però fare una considerazione. Nel momento in cui si leggono i pacchetti direttamente dal buffer, si può ragionevolmente ritenere che i tempi siano piuttosto regolari: si ha un ciclo che coinvolge sostanzialmente il solo processore e la memoria. Difatti, il grafico delle dispersioni nelle zone di coalescenza ha un andamento pressoché (in questo caso, *esattamente*) orizzontale. Tracciando su un grafico la *variazione tra le dispersioni successive* si ottengono valori molto piccoli, a limite nulli, all'interno delle zone di coalescenza. Comunque, si ottengono valori *indipendenti dalla velocità della CPU*.

In definitiva, estraiamo due *features*, "caratteristiche", dalle misurazioni di ciascun treno: la dispersione e la variazione delle dispersioni. Per prima cosa, viene comparata la dispersione con il rate a cui è impostata *attualmente* l'interfaccia, secondo un *criterio esclusivo*: se è inferiore a quella che si avrebbe alla velocità massima, allora decretiamo che *c'è* coalescenza (ed *escludiamo* quel tratto). Se invece così non è, allora la dispersione viene confrontata con la $\delta_{\text{k-u}}$ (moltiplicata per un certo k) secondo un *criterio inclusivo*: se è superiore di k volte allora certamente *non c'è* coalescenza. Se entrambi questi test non riescono a discriminare, allora ci rifacciamo alla variazione delle dispersioni. Sperimentalmente abbiamo visto che un buon termine col quale confrontarle è la kernel-to-user latency: se la variazione delle dispersioni è inferiore alla $\delta_{\text{k-u}}$ allora decretiamo che *c'è* coalescenza (*criterio esclusivo*).[8]



3.11/29(34)

Quindi, ricapitolando, operiamo per round. Alla fine di ciascun round effettuiamo i calcoli, estraendo le capacità, le feature, eccetera, e cerchiamo di stimare la capacità del percorso. Se per un certo numero di volte consecutive la capacità ricade entro i valori ricavati al passo precedente, allora ci interrompiamo: l'algoritmo si è stabilizzato. Inseriamo anche dei limiti massimi, entro i quali l'applicazione si ferma comunque.



3.12/30(35)

[8] Calcoliamo la kernel-to-user latency solo una volta all'inizio dell'esecuzione del receiver: possiamo infatti ricavarla in un modo che, con buona approssimazione, ci permette di tenere in conto possibili variazioni senza doverla necessariamente ricalcolare a ogni round (e senza dover assumere che la velocità della CPU rimanga costante durante l'intera esecuzione dell'applicazione). Semplicemente, inviamo e riceviamo in loopback un campione di 400 pacchetti (sempre di 1500 byte), appiccicandogli i timestamp, ne ordiniamo i valori ed escludiamo l'ultimo decile. È molto probabile che nel campione calcolato siano finiti i diversi valori che incontreremo in seguito: ne selezioniamo quello massimo, eliminando solo dei possibili *outlier*.