«بسم الله الرحمن الرحیم»

استاد: جناب آقای دکتر صبایی

درس: معماری افزاره‌های شبکه

جلسه 1

6/27

- تمرکز درس: طراحی روتر و سوییچ

- دیدگاه طراحی و پیاده‌سازی سوییچ‌ها:

○ در شبکه‌های سرعت بالا: { 1) لینک‌های سرعت بالا می‌خواهیم
2) سوییچ‌هایی که با سرعت بالا pkt switching کنند.

○ در گذشته: مشکل عمده، داشتن لینک سرعت بالا بود

○ اواخر دهه 80: فیبر نوری اومد ← مسأله شد سرییصینگ. (تا گلوگاه نشه)

⇐ مسأله‌ی درس: چه معماری‌هایی به کار ببریم که به سرییصینگ سرعت بالا برسیم؟

- یک روتر 2 دسته وظایف دارد:

1) control plane: عمل کنترلی ← مسیریابی ← به توپولوژی کل شبکه برمی‌گرده ← جداول forwarding را به روز می‌کند. (خروجی)

↳ mgmt plane

2) data plane: یه بسته که وارد می‌شه، از پورت درست خارج بشه. (انتقال داده) ← براساس forwarding data.

↳ در واقع یه forwarding engine داریم که عمل انتقال بسته از پورت ورودی به خروجی را انجام می‌ده و سرعت بالا خیلی مهمه.

↳ forwarding engine باید با سرعت خط کار کنه.

- وظیفه گره‌های میانی شبکه (نه گره نهایی): یه بسته از یه پورت میاد، یه کاری انجام می‌شه از یه پورت می‌ره. می‌تونه فایروال یا هر چی باشه.

○ در گذشته دیوایس جداگانه می‌خریدیم (روتر، فایروال و ...).

○ امروزه با SDN و NFV، اینا رو نرم‌افزاری انجام می‌دهیم.

- مراجع ← اسلایدها

سرفصل‌ها:

○ وظایف control plane: IP adds. rule - pkt classification - traffic mgmt ...

○ در بخش data plane: اصول pkt switching، سوییچ‌های shared memory - سوییچ‌های با بافر ورودی و ... را توضیح می‌دهند.

Scanned with CamScanner

- شبکه‌ی اینترنت، توسط یه سری شرکت های سرویس داده می‌شه. اپراتور می‌شه اینایی که سرویس می‌ده مثل همراه اول.

- اپراتور ها به 3 گروه تقسیم می‌شه:

1) tire 1   2) tire 2   3) tire 3

← مالک شبکه هستن. (فیبر و تجهیزات می‌ذارن.) مثال زیرساخت مخابرات، AT&T و ... . اینا با کاربر نهایی در ارتباط نیستن. سرویس می‌دهند به:

← tire 2 : شرکت های بزرگی که عمدتاً مالک شبکه نیستن، بهش‌ای باند اجاره می‌کنن. شرکت های بزرگ هستن که کاربر خرد ندارن. کاربر بزرگ دارن. مثل شرکت مخابرات ایران که از شرکت زیرساخت سرویس می‌گیرد. تعداد این 2 گروه خیلی زیاد نیست.

← tire 3 : سرویس محلی می‌دهند. مثل شاتل و ... ← تعداد زیاد. گاهی در یک محل خاص سرویس خوب می‌دهند. (از tire 2 سرویس اجاره می‌کنن.)

- اسلاید 2 توپولوژی: tire 1 : Core Network

نقاط حضور

○ به اون نقاطی که سرویس را فراهم می‌کنه point of presence (PoP) می‌گویند.

○ در PoP، یه زیرساخت (backbone) درست کرده سرویس می‌ده.

○ در PoP ها یه شبکه داره که یه سری روتر edge داره که به روترهای سطح پایین‌تر سرویس می‌ده. و یه سری هم روتر core. بحث peering هم داریم (با روترهای هم سطح خودش هم در ارتباطه)

- شکل اسلاید: 2 تا توپولوژی tire 1 تا 2 را نشون داده. چه تفاوت هایی دارن؟

1) full mesh: تعداد حضور نقاط کم ولی نودها همه به هم وصل هستن.

2) نقاط حضور زیاد ولی با چند گام گره های core به هم می‌رسند.

- ISP ها دارن کار تجاری می‌کنن (یه business دارن). پس توپولوژی با بازدهی بهتر می‌خواهند. یعنی سرویس بیشتر بتونه بده، هزینه نگهداری هم کمتر باشه.

- هزینه ها: 1) CAPEX : هزینه سرمایه‌گذاری

2) OPEX : هزینه نگهداری

← نمی‌شه جنس ارزون بخریم ولی برای نگهداریش کلی هزینه کنیم. برعکس هم نده چون بالاخره تکنولوژی منسوخ می‌شه.

⇐ مثلاً برای یک پروژه 10 ساله، هزینه capex + opex را حساب می‌کنن، کمتره را انتخاب می‌کنند.

Scanned with CamScanner

- در این 2 توپولوژی، هزینه متفاوته. کدوم بهتره؟

توپولوژی Sprint بیشتر مورد استفاده است. هزینه operation ← AT&T خیلی بالا می‌شه.

توپولوژی Sprint هزینه capex بالایی داره ولی هزینه نگهداریش کمتره. چون اولاً تعداد نقاطش کمتره. هر نقطه حضور، ساختمان و نگهبان و ... داره. فقط هزینه تجهیزات نیست. ⇐ هرچی PoP ها زیاد می‌شه هزینه‌ی open می‌ره بالا.

⇐ اکثر ISP ها تمایل دارن نقاط حضور را کم کنن ولی full mesh باشه. تنوع سرویس در نقاط حضورشون را زیاد می‌کنن ولی تعداد نقاط حضور را کم می‌کنن.

⇐ نقاط حضور با ظرفیت بالا نیازه. تعداد لینک‌های ورودی هم زیاد و متنوع است. ⇐ نقاط پیچیده می‌شه.

- اسلایدِ روترها: دو تا شرکت معروف Cisco و Juniper (در فایروال هم خیلی قوی هستن.) mid class ها برای شبکه‌های با اندازه متوسط - carrier class ها برای اپراتورهای با ظرفیت بالا.

- اسلایدِ نقاط حضور: 2 جور نقاط را می‌شه پیاده‌سازی کرد:

1) روترهای mid range را Cluster کنیم، از دید بیرونی ظرفیت بالا میشه. (خود اینها یه شبکه هستن)

2) یه روتر با ظرفیت بالا (پورت زیاد، قدرت سوییچینگ بالا) بذاریم.

کدوم بهتره؟

- از نظر هزینه: روترهای متوسط هزینه کمتر دارن. (10 تا بیاید، قیمت یک نبز هم نمی‌شه!) ← Capex کمتر
- از نظر توسعه و نگهداری: یک روتر با ظرفیت بهتره. چون scalable است (یعنی می‌شه با ظرفیت کم شروع کرد، هر وقت لازم بود، کارت interface اضافه کرد یا کارت سوییچینگ و ظرفیت را برد بالا.) ولی تو اون یکی باید روتر بخری، هزینه‌کنی لینک‌ها را وصل کنی و ...
- از نظر operation، مثلاً روتر 12000 به جای 7600 می‌ذاری، IOS فرق می‌کنه، کامندها ممکنه فرق کنه، دوباره هزینه کنی تا نگهداری کنی. (هزینه آموزش و غیره هم هست.) ← هزینه نگهداری یک نبز خیلی کمتر از 10 بیاید. اولاً دیر خراب می‌شه، بعد هم یه متخصص لازمه. اگه بیاید مدل مختلف باشه در زمان فرق می‌کنه.

⇐ اکثر ISP ها به روتر ظرفیت بالا تمایل دارند حتی ممکنه سفارش بدهند برایشون روتر بسازه.

- scalable: یعنی با ظرفیت کم هم می‌تونه کار کنه، ظرفیت می‌ره بالا فقط کارت اضافه می‌کنی. (مثلاً رم کامپیوتر را عوض کنی هیچ چیز تغییر نمی‌کنه.)

⇐ تمرکز درس روی روترهای با ظرفیت بالا هست.

Scanned with CamScanner

- مسائل امنیتی هم می‌تونن جزو چالش‌های طراحی باشن.

- وظایف روتر: 1) Data plane : شامل forward کردن بسته‌ها و تصمیم‌گیری اینکه از کدوم پورت بره. ← همون forwarding engine هایی هستن که عمل forwarding انجام می‌دهند.

2) Control plane :

- مسیریابی و تولید جدول forwarding
- کپ‌های مدیریتی

- به طور کلی معماری یک مسیریاب می‌تونه: متمرکز یا توزیع شده باشه.
  - وظایف control plane که متمرکز باید باشه (مدیریتی)
  - data plane می‌تونه توزیع شده هم باشه.

← در معماری a. pkt ها از interface ها دریافت می‌شن، تصمیم‌گیری می‌شن توسط forwarding engine که از کدوم پورت برد بیرون. ممکنه برای افزایش سرعت، چندین forwarding engine موازی داشته باشیم و هی متمرکز باشه.

← در معماری توزیع شده، forwarding engine ها را در interface ها می‌گذارند. (براساس اینکه از چه پورتی اومده به چه پورتی می‌ره، ممکنه از switch fabric عبور کنه یا نه.) ← یک بسته، حداکثر یکبار از سمت اصلی switching عبور می‌کنه و از پورت خروجی خارج می‌شه.

← در متمرکز، بسته میاد forwarding engine تصمیم‌گیری می‌شه برمی‌گرده. ممکنه switching fabric، گلوگاه، سیستم در متمرکز. (switch fabric می‌تونه یه bus باشه یا یه شبکه روی یک چیپ.)

- اگر N پورت داشته باشیم که هرکدوم یک forwarding engine داره، به اینا "line card" هم می‌گویند.
به linecard یه بسته از طریق مدیای فیزیکی (مثلاً فیبر نوری) وارد می‌شه، امواج نوری دریافت (یا هرچی)، تبدیل باید بشه به رشته بیت. اون tranceiver کار لایه فیزیکی انجام می‌ده مثلاً امواج را تبدیل می‌کنه به bit stream. (demodulation)

- framer کار Data link را انجام می‌ده. از یه رشته بیت، pkt ها را استخراج می‌کنه.

- Net. processor همون forwarding engine است که در حافظش forwarding table را دارد، بسته‌ی وارده را بررسی می‌کنه که از کجا خارج بشه. بسته را می‌ده به switch fabric، از یه پورتی خارج بشه. سمت اصلی switching، همین switch fabric است. این عملاً داره switching می‌کنه. ادنا می‌گن بسته از کجا خارج بشه.

- بسته که می‌رسه به پورت خروجی، چون QoS هم، بسته‌ها میاد در بافر خروجی، حالا مکانیزم مدیریت ترافیک لازمه که کدوم بسته بره. ⇐ traffic manager، pkt scheduling، buffer mgmt
انجام می‌ده. (بسته به tag ← بسته که کدوم زودتر بره به آخیر صف.) ⇐

Scanned with CamScanner

کار traffic mgmt را انجام می‌ده که 2 تا وظیفه‌ی اصلی دارد:

1) pkt scheduling 2) buffer mgmt

← وقتی لینک خالی شد، بسته‌ی جدید باید ارسال بشه، از بافر نوبت کدوم بسته است. ← مکانیزم خروج

buffer mgmt ← اگه یه بسته اومد و بافر پر بود، کدوم بسته loss بشه؟ (بسته ورودی یا یکی از صف) ← مکانیزم ورود

- اسلایدِ line card : line card یه CPU مرکزی داره که با control route mgmt در ارتباطه. این forwarding engine را دریافت می‌کنه، data هایی که قراره رد و بدل کنه از طریق CPU مرکزی انجام می‌ده.

⇐ یه مسیریاب داریم که هر line card خودش یه forwarding engine داره، مدیای فیزیکیش مشخصه. (فیبر نوری یا وایرلس یا هرچی) ← مستقل است.

این میاد اضافه می‌شه به یه روتری که switch fabric داره. اینا می‌تونن ماجولار باشن.

هرکدوم از line card ها می‌تونه یه سرعت داشته باشه.

⇐ یه روتر با ظرفیت بالای scalable خواهیم داشت.

- خود switch fabric هم می‌تونه ماجولار باشه. مثلاً تا 16 تا پورت، یه switch fabric داشته باشیم، بیشتر شد، switch fabric اضافه کنیم. (تعداد پورت زیاد می‌شه، ظرفیت switch fabric را هم اضافه کنیم.)

جلسه 2 — 6/29

مرور: 3 سطح ISP داریم:
1) مالک شبکه
2) اونایی که بهشون باند زیاد از مالک شبکه اجاره می‌کنن.
3) سرویس محلی به کاربران می‌دهند.

- ISP ها علامت دادن تعداد نقاط حضورشون را کم کنن ولی توپولوژی full mesh داشته باشن تا هم هزینه نگهداری را بیارن پایین و هم تنوع سرویس داشته باشن.

⇐ برای داشتن شبکه با سرعت بالا، لازمه روتر و سوییچ با ظرفیت بالا (large capacity) ① داشته باشیم که تنوع پورت داشته باشه و scalable ② باشه. (یعنی می‌شه با هزینه کم و ظرفیت کم خرید ولی بازمان که نیاز افزایش پیدا می‌کنه، اینو توسعه بدیم بدون اینکه تغییرش بدهیم.) ← مثلاً line card بهش اضافه کنیم.

- وظایف روتر: 1) data path ← pkt از چه پورتی خارج بشه، scheduling، buffer mgmt ...

2) control plane ← وظایف data path را کنترل می‌کنه. مثلاً به روز کردن جداول forwarding و rule ها و ... ← نقش تصمیم‌گیری و سیاست‌گذاری داره، data path اجراش می‌کنه.

maxnote
Scanned with CamScanner

- بخش data path وظیفه بالا می‌خواد در PoP ها. ← کارها ساده ولی حجم زیاد. در control plane، پیچیده است ولی تعداد کمتری دارد. ⇐ عموماً control plane، متمرکز پیاده‌سازی می‌شه ولی برای data path می‌ریم سراغ معماری‌های توزیع شده.

- معماری روترها: a) متمرکز: کارت‌های interface ها را داریم، forwarding engine ها هم پردازش روی بسته‌ها را انجام می‌ده (از چه پورتی خارج بشه، چه سیاست‌هایی رو بسته اعمال بشه.) می‌شه چند تا forwarding engine موازی داشته باشیم ولی به هرحال متمرکز هستند.

b) توزیع شده: به ازای هر پورت ورودی یه forwarding engine داشته باشیم. مستقل از هم توزیع شده کار می‌کنن. انتقال از پورت ورودی به پورت خروجی از طریق switch fabric انجام می‌شه.

- اون کارت‌هایی که interface های ما هستن و forwarding engine هم داخلشون هست، بهش می‌گیم line card. درون line card که به یک مدیای فیزیکی متصل هست، امواجی که روی اون مدیا هست را دریافت می‌کنیم. (مثلاً ممکنه فیبر نوری باشه یا کابل کواکسیال باشه. یه روش line coding یا مدلاسیون دارد.)

- در این tranceiver، عملیات لایه فیزیکی انجام می‌شه. یعنی امواج تبدیل می‌شه به رشته بیت. بعد از این، رشته بیت‌های صفر و یک داریم.

- framer وظیفه‌ی لایه‌ی data link را برای ما انجام می‌ده. از رشته بیت‌های دریافتی، pkt ها را استخراج می‌کنه. چون frame بندی کردیم. پس خروجی framer، packet می‌باشه.

- حالا pkt های IP داریم باید پردازش کنیم. یه network processor داریم برای پردازش لایه‌ی IP. چه عملیاتی؟ اول باید IP checksum هدر را چک کنه خطا نباشه، فیلد TTL را یک واحد کم کنه، اگه صفر شده، بسته را discard کنه اگه نه، بسته را پردازش کنه، آدرس مقصد را از روی بسته بردارد، به forwarding table مراجعه کنه، ببینه گام بعدی چیه، شماره پورت خروجی را پیدا کنه و پردازش‌های دیگه،

- بسته را به switch fabric بدهد، از پورت یک وارد شده، مثلاً از پورت k باید خارج بشه.

switch fabric وظیفش اینه که این عمل switching را انجام بده. یعنی هر بسته‌ای که میاد، توپولوژی مشخصه، معلومه از چه پورت خروجی باید خارج بشه. ورودی یه tag (lable) می‌زنه ← شماره پورت خروجی. switch fabric فقط براساس اون شماره پورت خروجی کار می‌کنه. کاری با محتوا و انیا نداره. هر بسته که اومده را switch می‌کنه.

- بسته وقتی می‌رسه به پورت خروجی، میاد در بافر خروجی در line card قرار می‌گیره. اونجا مکانیسم‌های pkt scheduling (نوبت‌بندی براساس اولویت و کلاس ← سیاست‌گذاری که انجام شده) و buffer mgmt اگه قرار بسته‌ای حذف بشه، کدوم حذف بشه. (براساس سیاست‌گذاری، اون که به QoS اش لطمه نمی‌خوره)

Scanned with CamScanner

این traffic manager نقش اعمال ~~[illegible]~~ (مکانیسم های) مدیریت ترافیک را انجام می ده در line card.

- route controller & mgmt که در control plane هستند به صورت متمرکز اجرا می شوند. پروتکل های مسیریابی که جداول forwarding را به روز می کنند، اونجا اجرا می شوند. (مثل OSPF و ...) نتیجه ی خروجی آنها، به روز کردن forwarding table ها است. اونا به روز می شه می ره به line card. یه CPU داریم که در ارتباط با control plane، communicate می کنه و داده ها را رد و بدل می کنه، این forwarding table یا سیاست های دیگه را به روز می کنه.

- عمدهٔ درس، همین معماری است. چه کارهایی در این line card ها انجام می شه، چه کارهایی در control plane انجام می شه و switch fabric چه طوری پیاده سازی می شه که بتونه در سرعت های بالا عمل switching را با n تا پورت ورودی و n تا پورت خروجی انجام بده.
← اینا گلوگاه های ما هستن. اگه کند عمل کنند، سرعت switching میاد پایین.
در هرکدام یه پردازش هایی باید بکنیم که زمان بر هستند. چالش عمده، پیاده سازی این هاست.

- ابتدای درس، تمرکز می کنیم روی control plane و data path. بعد تمرکز میاد روی switching. متعاقبش، بگیم چالش ها و مشکلات در طراحی چیا هستند که باید در نظر داشته باشیم.

- (اسلاید): چند نمونه switch ی واقعی. این switch ها چون باید scalable باشند و large capacity باشند، عموماً به صورت multi rack طراحی می شوند. (یه قفسه با چند shelf (طبقه))، روترها یه backplane (یه صفحه ای) دارن که با فیبرهای نوری به هم وصل شدن و یه ظرفیت بالای ارتباطی بین این slot ها درست کردن. کارت ها میان در این slot ها قرار می گیرن. با مثلاً 2 تا دونه line card کار می کنه تا مثلاً 100 تا.

- قسمت switch fabric هم باز slot بندی است. با یه دونه بورد switch fabric هم می تونه کار کنه (بستگی به تعداد line card ها و ظرفیت switching مورد نیاز)، ظرفیت switching را می خواهیم ببریم بالا، کارت های switching را افزایش می دهیم.
از نظر فیزیکی، روترهای با ظرفیت بالا، به این شکل طراحی می شوند که scalable باشند.

- 2 تا نمونه ی واقعی، روتر Juniper T640 Tx هست که شکل آن در اسلاید هست.
روتر Cisco CRS1 هم هست که این هم multirack هست.

- از نظر معماری، اون سمت Juniper، اون سمت -- TX matrice باید وجود داشته باشه. اون سمت مرکزی اصلی است. با یه دونه از اون T640 routing node ها می‌تونه کار کنه. (64 تا از این node ها می‌تونه بهش اضافه بشه.) هرکدوم از T640 ها خودش یه روتر کامل است که داخلش switching table forwarding و همه چی داره. ولی control plane ش متمرکز شده.

هرکدوم از نود ها را اگه باز کنیم، بسته ها را دریافت می‌کنه، داخلش بتدادی کارت switch fabric داره، routing or forwarding engine داره که از اون سمت اصلی داره می‌گیره. (control plane متمرکز است)

- روتر CRS1، یه مقدار معماری متفاوت‌تر داره. کارت های مستقل که داخلش switch fabric هست نداره. switch fabric را متمرکز پیاده کرده ولی از switch fabric های با معماری multi - stage استفاده کرده که با یه کارت کار می‌کنه. 8 تا کارت جداگانه می‌خوره. تا سرعت های 100 Gbps سوییچینگ می‌کنه.

هرکدوم از این کارت های switching، یک سوییچ کامل است. می‌تونه اینارو موازی کنه، بسته هایی که میان بین اینا پخش بشه که سرعت بره بالا.

line card ها هم که بهش متصل می‌شه.

- وقتی می‌خواهیم روتر با ظرفیت بالا طراحی کنیم، اول باید بگیم اهداف چیه؟ عمده‌ی اهداف برای طراحی روتر با ظرفیت بالا: (شبیه requirement analysis)

1) pkt forwarding performance: چند تا pkt در ثانیه را می‌تونیم سوییچ کنیم خارج کنیم؟ سرعت switching یا چند بیت بر ثانیه. ← هدف مهم ظرفیت.

2) scalability: با چی ظرفیت اولیه می‌تونه شروع به کار کنه، حداکثر به کجا می‌رسه. (با اضافه کردن ماجول ها)

3) bandwidth density: چگالی و تنوع پورت ها چی باشه؟ مثلاً سرعت های 10 kbps، ...، با چه مدیا های مختلفی؟

4) service delivery features: ویژگی هایی که می‌خواهیم داشته باشیم؟ مثلاً QOS می‌خواهیم؟ MPLS می‌خواهیم؟ (بیشتر از نظر کیفیت سرویس)

5) availability: بحث دسترس پذیری و قابلیت اطمینان. این روتر کجا قراره استفاده بشه؟ در core (مثلاً در tier 1 یا tier 2) که می‌خواد سرویس با ظرفیت بالا

Scanned with CamScanner

ارائه بده ⇐ شاید یک دقیقه down time هم برای ما قابل تحمل نباشه چون میلیاردها بسته
ممکنه از دست بره ⇐ availability بالا می‌خواد. مثلاً اگه احتمال می‌دهیم power بسوزه،
duplicate power باشه. مکانیزم های fault tolerance داشته باشیم که Single point of failure
نشه.

6) Security : اگه یه اختلال کوچک در forwarding engine ها به وجود بیاد، چون سرعت انتقال بالاست،
صدمه‌ی زیادی می‌خوره. اگه دسترسی غیرمجاز وجود داشته باشه که عمداً تغییری ایجاد کنه،
(availability بیشتر بحث خرابی سخت افزاری یا باگ های نرم افزاری است که غیرعمد
هستن. یه خرابی اتفاق می‌افته یا در برنامه یه باگ بوده.) ← کدها بالاخره باگ دارن، در
شرایطی خودش رو نشون می‌ده. این جا هم هزینه‌ی (نه پروتکل ها) debug کردن خیلی بالاست.
ولی بحث امنیت بحث دسترسی های غیرمجاز یا عمدی است برای ایجاد اختلال.

- قبل از طراحی، باید برای همه‌ی این ها معیارکمی تعریف کنیم، بعد در طراحی و پیاده سازی رعایت کنیم.
حالا این اهداف را داریم. ممکنه یه sys با مشخصات مورد نظر ما گردن باشه، یه trade off می‌شه بین نیازها و
قیمت. نیازها قیمت را تعریف نمی‌کنه. مثلاً می‌گن ما یه نرم افزار می‌خواهیم با فلان ویژگی ها. ما تحلیل می‌کنیم
می‌گیم می‌شه x تومان. اگه برات نصرفه، می‌گیم نیازها تو کم کن.

- برای طراحی، این چالش ها را داریم:
1) سرعت حافظه ها: طبق قانون مور، هر 18 ماه، سرعت پردازنده ها 2 برابر می‌شه. در memory، هر
12 ماه حجم حافظه ها 2 برابر شده. یعنی رشد، بیشتر از پردازنده ها بوده. (PC استاد، 30 سال پیش، یک
مگ رم داشت و 20 مگ هارد!)
مشکل اینه حجم در 12 ماه 2 برابر شده، ولی سرعت در 12 ماه 2 برابر نشده. پس سرعت پردازنده، در طراحی
دردسرساز می‌شه.
مثال: فرض کن یک line card داریم، یک خط ارتباطی ethernet داره با سرعت 10Gbps. می‌خواهیم
بسته هایی که میاد پردازش کنیم. بسته های IP مثلاً 20 بایت هدر IP هست و 20 بایت هدر TCP (لایه
UDP). مثلاً فرض کنیم بسته ها 100 بایت باشند. (800 بیت) ← اصلاً بگو 1000 بیت. ⇐ $\frac{10\times10^9}{1000}$ ⇐ 10 میلیون
بسته در ثانیه وارد می‌شه. اگه بخوار گلوگاه نباشه، باید با همین سرعت هم خارج بشه. ممکنه تأخیر بینهایت ولی
سرعتش باید یکی باشه. اگه سرعت خروجی کمتر باشه، یه سری loss می‌شوند. ⇐ هر بسته را باید در 100 nS
پردازش کنه. (یا 0.1μsec). اگه سرعت دسترسی به حافظه 10nS باشه، 10 بار می‌توانیم به حافظه مراجعه کنیم.
(fetch, execute, forwarding, search in table و پیدا کردن آدرس مقصد و... . تازه مکانیزم های کیفیت سرویس
هم هست.) 10 بار مراجعه به حافظه خیلی کوچکه. ← یکی از چالش ها همینه که الگوریتم هایی باید طراحی کنیم

Scanned with CamScanner

که تعداد مراجعه به حافظه ی کمتری داشته باشند. ⟸ یکی از چالش ها اینه که چون سرعت حافظه کند است، خودش مانع اصلی بالا بردن سرعت پردازش است ⟸ باید کمتر به حافظه مراجعه کنی (یا ساختار داده ی پیچیده استفاده کنیم که کمتر به حافظه مراجعه کنیم یا بخشی از عملیات پردازشی را بیاریم در سخت افزار. وگرنه 15 بار مراجعه به حافظه با اجرای 2 یا 3 تا instruction تموم می شه.)

2) چالش packet arbitration : فرض کن یه سوییچ فابریک داریم که n پورت ورودی دارد و n تا خروجی (داده‌ی کردن ↩) اگر (چون اینا موازی و توزیع شده دارن کار می‌کنن) همزمان بیش از یک pkt برسه به switch fabric که بخواد از یه پورت مشخص (j) خارج شه، کدوم را به پورت j بفرستیم. (اگه همزمان فقط یکی بتونه بره). یکی را باید بفرستیم، یکی را در حافظه نگه داری. پس بین اینا باید داوری کنیم (تصمیم بگیریم) به کدوم سرویس بدهیم. ← arbitration یا scheduling.

اگه n تا پورت داشته باشه، الگوریتم های scheduling که ساده هستن، در بهترین حالت از مرتبه‌ی n^3 هستند. یعنی در مرتبه‌ی n^3 عملیات باید انجام دهیم تا تصمیم بگیریم اول کدوم را بفرستیم ← اگه بخواهیم بهینه عمل کنیم. اگه مثلاً 100 پورت داشته باشیم، باید یک میلیون عملیات انجام دهیم که بگیم کدوم! به ازای هر سلول هم باید این کار را انجام دهیم! پس خود arbitration یه چالشه. (مجبوریم بهینه عمل نکنیم چون زمان انقدر نداریم.)

3) QOS : سلول ها بیان در بافرهای خروجی قرار می‌گیرند، ما می‌خواهیم بین اینها buffer mgmt / pkt sheduling انجام دهیم. k تا سلول در بافر هست، بین این k تا باید انتخاب کنیم که اگه خط خالی شه، سلول بعدی کدوم ارسال شه. یا اگه سلول ای وارد شد که صف پره، کدوم loss بشه که این جایگزینش شه. (به ترتیب scheduling / نوبت بندی ارسال ↩ buffer mgmt) ← سیاست گذاری ورود سلول ها به صف خود اینها پردازش لازم داره. این هم یه چالشه.

(داده)

4) optical interconnecting : اون backplane که در rack ها با فیبرهای نوری به هم وصل هستن، اینا اتصالات متعددی دارن. خود این اتصالات یک چالش است. هم انرژی زیاد (Power consumption) هم نحوه‌ی اتصالشون مشکله. ← پیچیدگی طراحی نیز یکی

5) Power Consumption : مصرف انرژی در روترهای multi rack خیلی بالاست (در حد چند ده هزار وات.) یکی از چالش ها، مصرف توان است.)

6) flexibility یا انعطاف پذیری : در نیازمندی ها، یکی از نیازمندی ها. service delivery feature یا ویژگی های تحویل سرویس بود که مثلاً چیارو پشتیبانی می‌کنیم ⟸ باید انعطاف پذیری داشته باشیم در ارائه‌ی سرویس. برای انعطاف پذیری بالا، باید پردازش را ببریم در نرم افزار که با یه تغییر کانفیگ،

Scanned with CamScanner

می‌توانیم شیوه‌ی سرویس‌دهی را عوض کنیم. ولی وقتی به خاطر بحث‌های حافظه و پردازش‌های پیچیده و محدودیت زمان، بیشتر کارها را ببریم در سخت‌افزار، انعطاف‌پذیری را از دست می‌دهیم. reconfig کردن سخت‌افزار هم آرسته ممکنه ولی خیلی محدودیت داره. هرچی می‌خواهیم سرعت را ببریم بالا، انعطاف‌پذیری را در ارائه‌ی سرویس از دست می‌دهیم. ⇐ یه tradeoff داریم. ← تا کجا انعطاف‌پذیری را داشته باشیم.

- کلاً وقتی می‌خواهیم کارایی یک sys بره بالا، 1) ساده می‌کنیمش 2) سخت‌افزاری پیاده می‌کنیم ⇐ سرعت بالا

وقتی می‌خواهیم قابلیت‌های متنوع داشته باشه، در سخت‌افزار نمی‌شه پیاده کرد. ولی سرعت را از دست می‌دهیم.

- در معماری، یک mgmt controller داریم، یک route controller. کنترل‌مسیرها، همان پروتکل‌های مسیریابی هستند مثل OSPF و غیره. ← تغییرات توپولوژی را می‌گیرن، براساس اون، بهترین مسیرها محاسبه می‌شه، جداول forwarding را به روز می‌کنن.

- در سمت mgmt controller :

در بحث مدیریت شبکه، ISO func. مدیریتی اصلی داریم که باید انجام دهیم : FCAPS

1) fault mgmt (مدیریت اشکال)

2) configuration mgmt (مدیریت پیکربندی)

3) accounting mgmt (مدیریت حسابداری و مالی) ← چه قدر از شبکه استفاده کرده؟ چه قدر هزینه بده؟ ⇐ باید اندازه‌گیری log شبه. صورتحساب صادر شه

4) Performance mgmt (مدیریت کارایی) ← مدیر شبکه باید دائماً کارایی را پایش کنه، اگه لازمه اقداماتی انجام بده.

5) Security mgmt (مدیریت امنیت) ← شامل 1) Protection (جلوگیری از تهدیدات امنیتی)

2) Detection (تشخیص حمله و از بین بردن اثراتش)

↳ بعد از حمله (مثلاً برگرداندن داده‌ها به کمک backup)

- مدیریت یعنی چی؟

هر sys، یه هدف و ماموریتی داره. برای این کار، ساختاری چیده می‌شه. (مثلاً دانشگاه، هدف آموزشی پژوهشی داره، براساس این هدف و ماموریت، دانشکده و مرکز رشد و پارک علم و فناوری و غیره داره) ← حالا همه‌ی اینارو درست کردیم، sys را روشن می‌کنیم کار کنه. آیا به اهدافش می‌رسه؟ خیر. چون در زمان طراحی، خیلی اتفاقات وجود دارد. (مثل نوع ورودی و غیره.) ← ما براساس یک کلیاتی طراحی می‌کنیم. مثلاً برای طراحی شبکه، نمی‌دونیم کاربران الان دقیقاً چه نیازی دارن. هدف اون که شبکه را ایجاد کرده (شرکت مخابراتی)، کسب درآمد است از طریق ارائه‌ی سرویس به مشتری‌ها. ⇐ هدف می‌شه ارائه‌ی سرویس خوب به مشتری‌ها. حالا در زمان، مثلاً یه اتفاق پیدا می‌کنه (مثل همراه اول و ایرانسل) ⇐ نیاز به مدیر هست.

نقش مدیر : sys را monitor بکنه براساس KPI هایی که تعریف می‌کنه، بعد ببینه در جهت اهداف

Scanned with CamScanner

میره یا خیر، آیا نه، کنترل کنه، تغییر بده. (مثلاً اگه یه اشتباه، متقاضی نداره، نیاز کشور هم نیست، تعطیلش کنه.)
یا مثلاً شرکت دائک لوند wimax ارائه بده ولی با ورود LTE و غیره، کنارش گذاشتن. ← مدیر تصمیم میگیره.
مدیر براساس چی تصمیم میگیره؟ داده های ورودی. ← نظارت کردن به شبکه، نیازهای مشتری ها، بررسی تکنولوژی های جدید. ← اطلاعات را میگیرد، پردازش میکنه، تصمیم مدیریتی میگیره، اعمال میکنه.
مثلاً معاون پژوهشی دانشگاه، باید علاوه بر نیازها، روش ها را هم بدونه (مثلاً ببینه دانشگاه های موفق چی کار کردن)، بهترین روش را انتخاب کنه، تصمیم بگیره. باید ابزار اجراش را هم داشته باشه.
← مدیریت شبکه هم یعنی، اطلاعات مدیریتی را دریافت کنیم (با monitor کردن شبکه، فرکانس، سرویس هایی که میدهیم چه قدر استفاده میشه؟ KPI ها چطوره؟ و ...) ← شاخص هم باید کمّی و قابل اندازه گیری باشه.
- monitor کردن sys، کفایت نمیکنه. باید نیاز مخاطبینش را هم درباره، با تکنولوژی های جدید هم آشنا باشه، براساس این 3 ورودی تصمیم بگیره تغییرات ایجاد کنه و کنترل کنه تا شبکه بتونه به اهدافش برسه. (مثلاً همراه اول هدف گذاشته 40 میلیون مشترک داشته باشه.)
⇐ مدیریت: پایش، monitoring، دریافت اطلاعات، پردازش، تصمیم گیری، اعمال کنترل.

- بخش monitoring را تجهیزات شبکه انجام میدهد، اطلاعات را جمع میکند. چه اطلاعاتی؟ اطلاعاتی که مدیر نیاز داره. اینارو باید براش تعریف کنیم. در غالب یه ساختار داده ی مدیریتی به اسم MIB تعریف میکنند. شرکت سازنده ی اون تجهیزات میاد در تجهیزات یک agent (عامل مدیریتی) میگذارد میگه براساس این ساختار، اطلاعات را جمع آوری کن. مدیر هر وقت بخواد ازش سؤال میکنه، اون agent اطلاعات را میفرسته.
⇐ ما اگر در پایین ترین سطح شبکه، تجهیزات شبکه را داشته باشیم ← network element (هر لاتر و سوییچ و غیره) ← داخل net. element ها یه agent مدیریتی داریم که معمولاً شرکت سازنده ایجادش میکنه.
وظیفه ی agent: براساس اون ساختار مدیریتی، اطلاعات را جمع آوری کنه.

- در شبکه مدیریت را سطح بندی میکنن:
1- element mgmt sys : مدیریت تجهیزات به تنهایی
2- network mgmt sys : مدیریت شبکه
3- مدیریت سرویس که شبکه ارائه میدهد. (service mgmt sys)
4- مدیریت کسب و کار (business mgmt sys)

(network element : تجهیزات تحت مدیریت)
- element mgmt sys، مدیر تک تک این هاست. نیز اگه میخوام اینو config کنم، کار اینه. مثل cisco works که با حاش میتونیم تجهیزات cisco را remote کانفیگ کنیم.

Scanned with CamScanner

- net. mgmt. sys. : شبکه را مدیریت می‌کنه. دسته تجهیزات به صورت مجزا دارن مدیریت می‌شوند ولی معلوم نیست کل شبکه درست کار بکنه. (مثل یک تیم) ← ممکنه تک تک تجهیزات درست کار کنه ولی سرویس خوب نده.

- در net mgmt sys به جزئیات تجهیزات کاری نداریم. اینکه ساخت Cisco است؟ یا ... اوتر، اوتره.
در element mgmt sys، خاص هر element است. مثلاً اگر Cisco باشه، یه جور کانفیگ می‌شه، Juniper یه جور دیگه.
پس element mgmt sys معمولاً خیلی وابسته است به شرکت سازنده، معمولاً هم خود شرکت سازنده سیستم مدیریتی‌اش را می‌ده. مثلاً Cisco، Cisco works را داده. ولی net mgmt sys به برند و اینا کاری نداره به وظیفه کار داره. مرکز مثلاً وظیفه‌ی اوتر، فلانه.

- در لایه‌ی service mgmt و business mgmt که معمولاً به اسم OSS ازش نام می‌برند، اینا از تجهیزات خارج شدن دارن سرویس را مدیریت می‌کنن و کسب و کار را. (بیشتر بحث های مالی و سرویس براشون مهمه)
مثلاً همراه اول میاد میگه تخفیف یک روزه دارم برای فلان. این به تجهیزات ربط نداره. بحث بزینسی است.
← بحث های مالی (billing mgmt) در همین بحث سرویس و business هستند.
- اینا همه sys های مدیریتی در سطوح مختلف هستند.

- اگه یک network element، یک اوتر را در نظر بگیریم، این network element داخلش یک agent مدیریتی داره که براساس اون mgmt information base یا MIB، اطلاعات را جمع آوری و نگهداری می‌کنه و در اختیار اون element mgmt sys می‌گذارد.
- برای این ارتباط باید یک پروتکل امنیتی تعریف کنیم که این اطلاعات ارتباطی را با یه پروتکل باید رد و بدل کنیم دیگه. در شبکه‌ی اینترنت از SNMP (simple net. mgmt protocol) استفاده می‌شه.

- پروتکل SNMP، اطلاعات مدیریتی‌اش را در غالب یک ساختار ساده بیان می‌کنه. (ساختار سلسله مراتبی)
اون object هایی که می‌خواد جمع آوری کنه هم، object های سادن، counter (شمارنده)، هستند و scalar (عددی) هستند. مثلاً تعداد pkt های lost شده، تعداد connection TCP های باز و ...
مثلاً ساختار MIB2 را سرچ کنیم. مثلاً گروه sys داره، گروه interface، گروه IP، گروه OSPF و ....
← گروه بندی کرده. در sys، اطلاعات sys میاد که مثلاً این sys در چه لایه‌ای کار می‌کنه؟ مثلاً اگه اوتر هست، می‌شه لایه 3. کجا نصب شده؟ uptime اش چه قدر بوده؟ ...
← در گروه interface ها: تعداد کارت هایی که توش هست، وضعیت کارت ها و ... با این می‌شه یک بندی هم کرد. مثلاً اگه یه interface، down هست، مدیر می‌تونه request بده به agent که این کارت را disable کن. (تغییر اون bit باعث می‌شه config عوض شه.)

Scanned with CamScanner

- پروتکل SNMP، یه پروتکل client-server هست که براساس request-response کار می‌کنه. (manager نقش client داره. ← مصرف کننده‌ی اطلاعات مدیریتی. agent: نقش تولیدکننده‌ی اطلاعاتی)

SNMP request ← response ←

- SNMP request ها اینطوریه: مدیر، object ID رو می‌ده. چون ساختار درختی است، ساختار naming هم داره، هر object، یک ID داره. object ID رو می‌ده (unique هم هست چون درختی چیده.) ← می‌گه این object ID رو می‌خوام get کنم. مقدارش رو agent برمی‌گردونه. یا می‌خوام set کنم (مقدارش رو عوض کنم.) response می‌ده که set انجام شد.

get, set، متدهای اصلی هستند. get-next هم داره. مثلاً می‌خواد جدول routing رو به دست بیاره، هی get-next می‌کنیم، object بعدی رو می‌گیریم، اطلاعات جدول رو از agent دریافت می‌کنیم. یعنی یه ID می‌دیم می‌گیم ID بعدی رو بده. چون اطلاعات جدولی است، پیمایش انجام می‌شه. MIB browser ها، با get-next می‌رونه جلو تا تموم شه.

- SNMP قابلیت دیگری داره به نام trap. یعنی manager از قبل به agent می‌گه اگه این اتفاق خاص افتاد، به من بگو. اتفاق خاص چیه؟ manager تعریف می‌کنه. مثلاً می‌گه این object (که معمولاً شمارنده است مثلاً تعداد pkt هایی که TTL = 0 شده، حذف شده)، می‌تونه threshold براش رو هر کدوم از counter ها بگیم از این th. رد شد، trap بفرست.

manager این اطلاعات را در DB ذخیره می‌کنه، پردازش می‌کنه، یه interface گرافیکی داره معمولاً که کاربر باهاش کار می‌کنه. (مثلاً اونایی که قطع می‌شه، قرمز می‌شه، یا جاهایی که ترافیک می‌ره بالا، رنگش عوض می‌شه.)
← بعضی از کارها از قبل تعریف می‌شه، مثلاً اگه فلان اتفاق افتاد، خودکار recover بشه ولی عمدتاً دستی تصمیم‌گیری می‌شه.

Scanned with CamScanner

نیمسال دوم جلسه 2 مجازی

slide: ip address lookup

(20 دقیقه آخر)

2 | مجازی

- به سراغ بخش forwarding engine می‌رویم.

- وظیفه‌ی اصلی forwarding engine :
وقتی یه بسته دریافت می‌شه، گام بعدی را مشخص کنه، pkt را از طریق بخش switching و نهایتاً پورت خروجی، ارسال کنه.

اسلاید : IP Adress Lookup ← جستجوی آدرس IP در جدول forwarding

⇒ وظیفه‌ی اصلی forwarding engine : تعیین شماره پورت خروجی برای فرستادن بسته به آن و خروج بسته از میزبان

P.3 : برای pkt forwarding ، یک بسته که دریافت می‌کنیم، این پردازش‌ها باید روی بسته انجام شود :
1) هدر بسته را decapsulate کنیم، فیلدها شو جدا کنیم.
2) TTL را update کنیم. (هر گام، یکی ازش کم بشه.) ← اگه TTL صفر شد، بسته loss می‌شه.
3) IP checksum را چک کنیم خطایی در بسته وجود نداشته باشه.
⋮
⇐ یه سری از این اقدامات داریم که هر کدوم زمان زیادی نمی‌خواهد شاید در حد یک instruction باشن. چون کارهای ساده‌ای هستن، می‌شه به صورت سخت افزاری در net. processor انجام بشه.

بخشی که زمان‌گیر است، اینه که در جدول forwarding باید جستجو کنیم برای IP مقصد و به دست آوریم، گام بعدی بسته برای رسیدن به مقصدش، کدوم گره‌ی مجاور هست و بسته را از اون پورت خارج کنیم به اون گره‌ی مجاور برسه.

اگه اندازه‌ی جدول forwarding ، به اندازه‌ی یک میلیون entry (آدرس) باشه، جستجو کردن برای پیدا کردن آدرس مقصد و next hop ، زمان زیادی می‌گیرد.

⇐ با اون وظایف اول که ساده بود، کاری نداریم اما زمان جستجو در جدول، پیچیده است، ضمناً محدودیت شدید داریم در زمان forwarding در حد چند ده نانوثانیه. ⇐ نیاز به روش‌های جستجو با توجه به performance مورد نیاز داریم. ⇐ یک چالش عمده : انتخاب روش جستجو در جدول forwarding

P.4 : با خود جستجو آشنا شویم : آدرس IP ، 32 بیت دارد.

maxnote

Scanned with CamScanner

P.5 : برای اینکه اندازه‌ی جدول forwarding کوچک باشد، ما در پروتکل اینترنت، آدرس IP را به صورت دو بخش Net. ID و Host ID تقسیم می‌کنیم.

P.6 : پس جدول forwarding که در هر گره داریم، شامل آدرس مقصد و next hop هست را داریم.

P.7 : از روش آدرس دهی سلسله مراتبی استفاده می‌کنیم.

P.8 : بخش آدرس را به 2 بخش Net ID و Host ID دسته‌بندی می‌کنیم. برای ما، شبکه‌ی مقصد مهم، پس در جدول forwarding، آدرس شبکه های مقصد را نگه می‌داریم و host ID را صفر می‌گذاریم. (برای ما مهم نیست.)

P.9 : یک نمونه از جدول forwarding.
وقتی یک بسته دریافت می‌کنیم که آدرس مقصد در هدر بسته وجود دارد، آدرس مقصد، یه عدد 32 بیتی است، اونو برمی‌داریم، مثلاً 20 بیت اول را با سطر اول (120) مقایسه می‌کنیم، با اون (116)، 16 بیت اول را مقایسه می‌کنیم و ... . یعنی در هر مقایسه فقط Net ID را مقایسه می‌کنیم.

- در عمل، برای پیاده سازی، اگر می‌خواهیم 20 بیت را مثلاً مقایسه کنیم، اون 12 بیت کم ارزش را باید از بین ببریم، می‌آییم با یک net. mask، and می‌کنیم که 20 تا بیت 1 و 12 بیت صفر داره. بعد مقایسه می‌کنیم. این طوری عملاً host ID را مقایسه نکرده ایم.

P.10 : قبل از سال 1993، شبکه های بزرگ، کلاس A بود و ... تا کلاس D. فیلدهای Net ID و host ID، fix بود. عملاً net mask را ذخیره نمی‌کردیم. از آدرس شبکه‌ی مقصد می‌فهمیدیم در چه کلاس قرار دارد.
اگر بیت اول 0 بود ← کلاس A (8 بیت اول Net ID) اگر 10 ← کلاس B (16)
اگر 110 ← کلاس C (24) اگر 1110 ← کلاس D

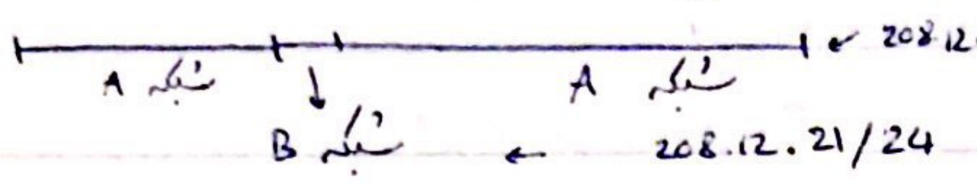
- اما چون تنوع شبکه ها بیش از این 3 کلاس بود، هدر رفت فضای آدرس وجود داشت، لذا در سال 1993، آدرس‌دهی classless شد. ← net mask هم همراه آدرس مقصد ذخیره می‌شد. (یعنی Net ID)

P.12 : اگر رنج آدرس ها را از (32 تا 0) تا (32 تا یک) در نظر بگیریم، می‌شه محدوده‌ی آدرس های IP V4. هر شبکه، یک محدوده آدرس می‌گیرد.

P.13 : ← P.11 اگر آدرس های پشت هم به یک شبکه داده باشیم، می‌توانیم address aggregation انجام دهیم.

Scanned with CamScanner

p.13 : یک مسأله به وجود آمد که ممکن بود، یک بخش آدرس را به یک شبکه‌ای دادیم (مثلاً شبکه‌ی A)، بعد یک استثنا وجود داشت که اونو به شبکه‌ی B داده بودیم :



دیگه نمی‌شد با address agg. همه‌ی اینارو ترکیب کنیم به طاش بگیم کلاً : 208.12.16/20

برای حلش، گفتن در address agg. ، این استثنا را در نظر نگیریم، بگیم : شبکه‌ی A : 208.12.16/20

ولی این غلط هم بود، چون آدرس مقصد شبکه B چی! ← مثلاً شبکه B که زیرمجموعه‌ای از شبکه‌ی A هست را هم در جدول forwarding ذخیره کن : شبکه‌ی B: 208.12.21/24

شبکه‌ی کوچکتر، Net ID بیشتری دارد. حالا اگه بسته‌ای با آدرس مقصد 208.12.21.110 بود، با اون مقایسه کن، 20 بیت اولش یکی است، با دومی، 24 بیت match می‌شه. الان با هر دو match شد، در این صورت می‌فرستیم به شبکه کوچکتره که prefix بیشتری دارد. ← longest prefix matching

⇐ در سرچ اگه یه match پیدا کرد، باز باید ادامه بده شاید یه match بهتری پیدا کنه.

p.14 : به این روش آدرس دهی، Classless Interdomain Routing یا CIDR می‌گویند.

آدرس های مقصد با 2 تا عدد 32 بیتی نمایش داده می‌شه، ← آدرس / net mask

مثلاً : 12.0.0.0/16 → 12.0.0.0 , netmask: 255.255.0.0

از روی هدر بسته، هر آدرسی را که برمی‌داریم، اول با net mask ، and می‌کنیم، مثلاً 12.0.54.2 می‌شه 12.0.0.0 ، که این بسته با 12.0.0.0/16 ، match شد. با 12.0.54/24 هم match می‌شه. پس متعلق به شبکه‌ی 12.0.54/24 است.

3 مجازی

3 مجازی

p.14 : دیدیم نوع جستجو LPM است. وقتی با یک مسیریاب core شبکه روبرو می‌شویم، این مسیریاب در جدول forwarding چه آدرس هایی را به طور معمول نگهداری می‌کند و این جدول چه ویژگی‌هایی خواهد داشت؟ ↓

p.15 : با توجه به نمودار، نمودار رشد اندازه‌ی جدول forwarding یک روتر backbone نشان داده شده از سال 1989 تا 2010. محور عمودی، تعداد entry های جدول forwarding است. همانطور که دیده می‌شود، رشد آن خیلی از حدش هم بیشتر است. ⇐ اگر فرض کنیم روشی را برای IP lookup پیشنهاد می‌دهیم، این روش باید با افزایش تعداد entry ها باز هم کارا باشد ⇐ باید Scalable یا مقیاس پذیر باشد.

p.16 : در این نمودار، لودن یه snapshot از روتر گرفتن، توزیع فراوانی prefix هاش رو کشیدن. (یعنی شبکه‌های مقصد که net. ID ، تعداد بیت‌هاش از 0 تا 32 باشه.)

زیر 8 بیت، هیچ آدرس مقصدی نبود.

محور عمودی نمودار، حالت لگاریتمی دارد.

Scanned with CamScanner

اکثر prefix ها طولی بین 16 تا 24 دارند، بیشترین در 24 بوده. ⇐ اگر فرض کنیم 90 درصد prefix ها 16 تا 24 بیت هستند، روش پیشنهادی باید این ها را با سرعت بهتری جواب دهد. ← تمرکز زمان پاسخ یا زمان جستجو باید روی این محدوده باشد تا کارایی بهتر باشد.

⇐ روش های ما باید: 1) scalable باشد. 2) زمان پاسخ برای prefix های 16 تا 24 خوب باشد.

p.17 : اهداف اصلی طراحی روش IP lookup :

1) کاهش زمان جستجو (به 2 زیرهدف تقسیم می شه):

a) تا حد امکان تعداد دسترسی به حافظه را کاهش دهیم.

b) اگر حجم کلی حافظه را کاهش دهیم، با توجه به اینکه access time در memory ها به حجم بستگی دارد، زمان دسترسی سریع تر می شه. ← از data structure ی استفاده کنیم که حجم کمتری بگیره.

2) اگر از روش نرم افزاری استفاده می کنیم، کد ها را کاهش دهیم. (تعداد instruction ها)

3) از data structure های با ساختار ثابت (aligned) استفاده کنیم. یعنی مثلاً اگر از ساختار درخت جستجو استفاده می کنیم، اندازه ی نود های درخت ساختار داده های ثابت با طول ثابت باشه. (مثلاً یه نود 15 بایتی نباشه یه نود 50 بایتی)

p.18 : در مورد جستجوی آدرس، اگر IPv4 باشه، جدول حداکثر می تونه 2^{32} entry داشته باشه. اگر اینو در غالب درخت جستجوی باینری نمایش دهیم، یک درخت با عمق 32 خواهیم داشت.

هر entry ی جدول (مثل e1 یا e2)، یک ناحیه ای را پوشش می دهد. اگر e2 زیرمجموعه ی e1 باشه، یعنی prefix طولانی تری داشته باشه، اون زیرمجموعه ی بعدی است. ما می خواهیم درخت جستجو را به روشی جستجو کنیم که LPM را انجام دهیم.

p.19 : یک ایده در بحث IP address lookup مطرح می شه که از تکنیک های caching استفاده کنیم. cache یک حافظه ی موقتی است هست که بین حافظه و پردازنده، قرار می گیرد. برای اینکه این حافظه حجم کمتر و access time کمتری دارد، سرعت اجرا را بالا می برد.

⇐ یک ایده مطرح می شه که بخشی از جدول forwarding را به حافظه ی cache منتقل کنیم. بعد به جای اینکه با حافظه کار کنیم، با cache کار کنیم. (چون access time کم می شه، سرعت جستجو زیاد می شه.)

Scanned with CamScanner

این ایده. از این جا بیاد که وقتی فرستنده می‌خواهد فایلی به گیرنده ارسال کند، یه Stream از pkt ها درست می‌شه (مثلاً 1000 تا بسته‌ی پشت هم.) => اگه یکبار lookup کنی، اطلاعات را بیاری روی cache، بقیه بسته‌های بعدی می‌شه به cache مراجعه کنی ← مراجعه‌ی محلی به data.

در واقع تکنیک caching زمانی کارایی دارد که ما به مراجعه‌ی محلی به حافظه داشته باشیم، یعنی وقتی که مراجعه بیش زیاد داریم را بیاریم روی cache. (زمانی که locality وجود داشته باشد.)

- اما وقتی اومدن تحقیق کردن در بخش core که روترهای با سرعت بالا هست، دیدن درسته به Stream از داده منتقل می‌شه که می‌خواد به مقصد برسه ولی همزمان، جریان‌های متعدد دیگری هم وجود دارد که داره از این لینک عبور می‌کنه. => بین هر دو pkt متوالی از یک جریان، میلیون‌ها pkt دیگه از جریان‌های دیگه وجود دارد و این ارجاع محلی در مسیریابی اتفاق نمی‌افتد. => چون locality لازم در گره‌های core وجود ندارد، caching روش خوبی نیست. => تمام روش‌های مبتنی بر cache برای IP lookup را کنار می‌گذاریم.

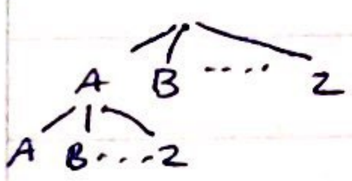
p.20 : وقتی می‌خواهیم یک روش IP address lookup را ارزیابی کنیم، باید ببینیم چه چیزهایی برای ما اهمیت دارد؟ (معیارها)

1) سرعت جستجو
2) حجم حافظه‌ی مصرفی
3) زمان به روزرسانی جدول forwarding
4) مقیاس پذیری
5) انعطاف پذیری (پیاده سازی) (flexibility)

p.21 : الگوریتم‌های IP addr. lookup را می‌توان به 2 دسته تقسیم کرد:

1) روش‌های نرم افزاری (مبتنی بر درخت جستجوی Trie)

این برای جستجو در DB های بزرگ استفاده می‌شه مثل فرهنگ لغات که از مثلاً 26 حرف شروع می‌شه، بعد حرف دوم دوباره یکی از 26 کاراکتر:



اگر بخواهیم مثلاً Help را جستجو کنیم، با 4 بار مراجعه به حافظه => معنی لغت پیدا می‌شه. (ممکنه هزاران کلمه در حافظه ذخیره کرده باشیم ولی با 4 بار مراجعه به حافظه پیدا می‌شه.)

2) روش‌های سخت افزاری : اگه زمان نداشته باشیم برنامه را اجرا کنیم و روش‌های با کارایی

maxnote
Scanned with CamScanner

بالایی بخواهیم، سراغ روش های سخت افزاری می‌رویم. ← الگوریتم صبحتر با مدارهای سخت افزاری پیاده می‌شه.
⇐ سرعت بالا می‌ره ولی انعطاف پذیری کاهش پیدا می‌کنه و update time افزایش پیدا می‌کنه.

⇐ براساس نیاز، بین این 2 روش، انتخاب می‌کنیم.

P.22 : روش های Trie - based

- binary Trie (درخت صبحتری باینری) یکی از روش ها
- path compressed Trie
- Multi-bit Trie (در مقابل binary)
⋮ ← بقیه در اسلاید از نوع multi-bit Trie

- یک ایده‌ای وجود دارد تحت عنوان tree bitmap algorithm ← درختی است که از یک نگاشت بهتر ایجاد می‌شود. جزو روش های multi-bit Trie است که با یک data structure ثابت، کارایی خوبی دارد.

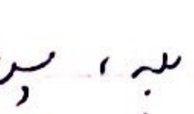
- استفاده از روش های pipeline برای افزایش سرعت.

- تعداد روش های مبتنی بر Trie خیلی زیاده، ما این روش‌های نام‌برده شده در اسلاید را می‌خوانیم.

P.23 : Binary Trie :

یک درخت binary است.

- اگر جدول forwarding را طبق اسلاید در نظر بگیریم، از ریشه که شروع می‌کنیم، هر گره، 2 تا لینک داره.
⇐ اگر بیت آدرس صفر باشه، لینک سمت چپ، اگر یک باشه، سمت راست.
⇐ اگر مسیری که وارد شده، مثلاً آدرس مقصدش باشه : 1 .... 11010 ، بیت اول، یک است، می‌ریم راست، دوباره 1، می‌ریم راست .... . انقدر ادامه می‌دهیم تا به برگ درخت می‌رسیم.

- درخت چطوری اجرا می‌شه؟ next hop را چطوری پیدا می‌کنیم؟ درخت از روی جدول forwarding ساخته می‌شه.
می‌آییم از ریشه شروع می‌کنیم. نگاه می‌کنیم آدرسی داریم با بیت 1 شروع شده؟ بله، پس گره‌ی  را ایجاد می‌کنیم. اگر *1 وجود داشته باشه، داخل این گره، next hop را ذخیره می‌کنیم:
همینطور ادامه می‌دهیم.
- در بعضی گره های binary Trie، next hop درشون ذخیره شده، در برخی نه.

Scanned with CamScanner

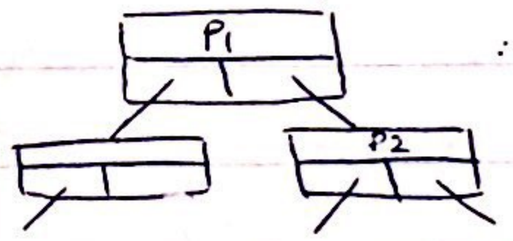
- هرچی به عمق می‌رویم، prefix های طولانی‌تری داریم. چه طوری پیمایش می‌کنیم؟ یه آدرس 32 بیتی میاد، از روی آدرس مقصد، بیت‌ها را برمی‌داریم درخت را پیمایش می‌کنیم.
مثلاً اگه آدرس مقصد 1... 110101 باشه، اول 1، می‌گیم next hop، a است، بعد 110 را می‌بینیم، تا این جا می‌گیم گام بعدی، d است. 1101، می‌گیم گام بعدی f است. دیگه نمی‌تونیم پیمایش کنیم => next hop می‌شه f.
=> آدرس 32 بیتی را نگاه می‌کنیم، درخت را پیمایش می‌کنیم تا جایی که نتونیم ادامه بدهیم. آخرین next hop که دیدیم همون LPM است.

- برای حذف کردن هم، اگه مثلاً h را بخواهیم حذف کنیم، h را که باید حذف کنیم، گره‌ی قبلیش هم که next hop توش نیست حذف می‌کنیم. بعد می‌ریم گره‌ی ماقبلش، چون next hop داره، حذف نمی‌کنیم.
اگه بعد h، c را هم حذف کنیم، می‌ره تا ریشه همه حذف می‌شن. ریشه را نمی‌توانیم حذف کنیم چون شاخه‌ی دیگه وجود دارد.

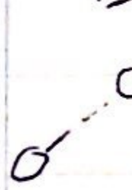
p.24: اگه بخواهیم این را پیاده کنیم، در هر گره، ما یه ساختار داده‌ی به شکل زیر نیاز داریم:

next hop info.	
L Pointer	R Pointer

که دوتا Pointer داره به سمت راست و چپ (با صفر و یک می‌ره)، داخل این گره، اطلاعات گام بعدی را نگه می‌داریم.
مثلاً:



p.25: این روش binary Trie، از نظر کارایی چه طوره؟
در بدترین حالت، برای یک lookup، اگه طول prefix، 32 بیت باشه، حداکثر 32 بار مراجعه می‌خواهیم تا next hop را بدست آوریم. اگه بخواهیم یک prefix جدید اضافه کنیم، در بدترین حالت، باید 32 تا نود اضافه کنیم.



← تعداد entry های جدول forwarding

- اگر اندازه‌ی جدول N باشه، 32xN، هر گره هم S بایت حافظه ببره، حجم حافظه‌ی مصرفی از مرتبه‌ی O(32N.S) می‌شه. البته بعضی از گره‌ها مشترک می‌شوند، پس یه کسری از این 32xN می‌شه. مثلاً 32xn.

- اگر prefix ما، طولانی‌ترینش w بیت باشه، (32 نباشه)، مرتبه‌ی جستجو می‌شه: O(w) → زمان جستجو و حافظه‌ی مصرفی هم می‌شه O(Nw). (اون S، یه عدد ثابت بود دیگه.)

← مثلاً در جدول ص 24: N = 9 , w = 7

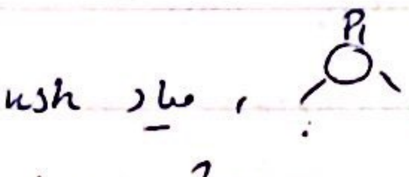
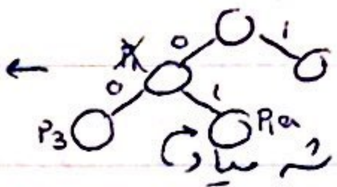
maxnote
Scanned with CamScanner

p.28 : یک بهبودهای داده شده به binary Trie ، مثلاً :

- Disjoint-prefix binary trie :

میاد می‌گه اون درخت binary trie (مثلاً ص 24) را کاری می‌کنه که گره های میانی‌اش ، اطلاعات next-hop را ذخیره نکنه بلکه این ها رو push می‌کنیم به سمت برگ ها . ⇐ فقط برگ های درخت next hop را نشون می‌دهند.
به این ترتیب اون ساختار داده ی (شامل next-hop information و دوتا پوینتر) ، گره های میانی دیگه next-hop نداره. گره های برگ هم چون pointer ندارند، می‌شه در یکی از pointer ها next-hop info را ذخیره کنند. ⇐ ساختار داده از نظر حجم کاهش میابد (حجم هر گره مثلاً 30 درصد کاهش میابد)

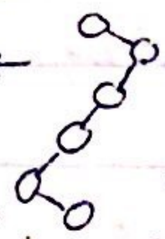
- چطوری این کار را انجام دهیم؟

در حالت اول ، P_1 اون بالا بود  ، میاد push می‌کنه پایین ، بعد P_1 که اون بالا بود هم پاک می‌کنه . (آدرس مقصد ، بیت اولش یا صفره یا یک ← P_1 ($P_1 = *$))
حالا دوباره ، P_1 را هول می‌ده .  ← الان *00 باشه ، P_3 است ، چون P_3 طولش بزرگتره ، P_1 دیگه این سمت نمیاد . ولی 01 باشه میاد

⇐ از ریشه شروع می‌کنیم ، prefix هایی که داریم هول می‌دیم به سمت پایین ، هرجا که طول کمتری داشته باشه .
در درخت هم گره های ماقبل انتهایی ، حتماً باید 2 تا برگ را داشته باشه .
درختی که درست می‌شه ، همه ی برگ هاش کامل است . ← درسته حجم هر گره را کاهش داریم ، اما مجبور شدیم یه سری گره اضافه کنیم نسبت به binary trie . موفقیت این روش مفیده که در مجموع ، حجم حافظه کاهش پیدا کنه .

- از نظر پیچیدگی ، با مقیاس یک order دارد .

p.27 : path-compressed trie

در درخت ص 24 ، مسیرهای طولانی داریم که شاخه های دیگری وجود ندارد . ما می‌توانیم اون شاخه های خلوت را که شاخه های زیادی ندارد ، (مثلاً  ← شاخه های زیادی ندارد) را فشرده کنیم . مثلاً از P6 می‌ریم به P9 اگه بیت بعدی صفر باشه . skip می‌کنه
یا از 2 تا بیت می‌پریم . (11) یعنی اگه در جستجو رسیدیم به ...011 ، مستقیم می‌ریم به P9 از P6 . ⇐ 2 بار مراجعه به حافظه را می‌توانیم کم کنیم (برای 11) .

⇐ path-compressed trie این کار را انجام می‌ده . مسیرهایی که شاخه ندارند را از نظر تعداد گام فشرده می‌کنه و یه تعداد بیت را skip می‌کنه .

Scanned with CamScanner

هر تعداد بیت را Skip کرده (Skip Value) و مقدار بیت‌های Skip شده هر برد (Segment) را در گره‌ی بعدی ذخیره می‌کند.

← از نظر ساختار داده، 2 تا فیلد به هر گره اضافه کرده ← و تعداد گره‌ها را کاهش داد.

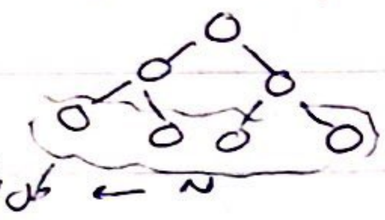
(در روش قبلی ساختار داده را کوچک کرد، تعداد گره را زیاد، این جا برعکس.)

← هر کدوم از این روش‌ها، براساس نوع جدول forwarding ممکنه اثر خودشو داشته باشه.

P.28 : در روش path-compressed trie ، اگر درخت جستجو یک درخت کامل باشه، هیچ بهبودی نخواهیم داشت. زمانی بهبود داریم که درخت خلوت باشه با شاخه‌های طولانی. این جا فشرده سازی مؤثره.

- در روش disjoint prefix binary trie ، اگر درخت کامل باشه، push کردن prefix ها به سمت برگ ها می‌تونه حجم حافظه را کاهش دهد.

P.29 : از نظر کارایی، دو تا روش همون order ـ قبلی را دارند. ($O(NW)$ نیست البته) اون s که حجم هر حافظه بود، افزایش پیدا می‌کنه و معلوم نیست در عمل حافظه‌ی کمتری مصرف کنه.



کل نودها می‌شه ضریبی از N ← N

- Multi-bit trie

P.30 : در روش Multi-bit trie ، به جای اینکه یک بیت را برای تقسیم گیری استفاده کنیم، چند بیت را برای تقسیم گیری استفاده می‌کنیم.

- مثلاً در ص 30 ، 3= در نظر گرفتیم، یعنی 3 بیت را در نظر گرفتیم برای تقسیم گیری. => اگه 3 بیت 3 بیت، بیت‌های مقصد را در نظر بگیریم، 8 تا pointer خواهیم داشت به گره های بعدی.

وقتی k بیت k بیت می‌ریم جلو و اگه تعداد بیت w باشه، تعداد پیمایش در جدول می‌شه $O(\frac{w}{k})$ یعنی زمان جستجو کاهش پیدا می‌کنه.

=> هر چه تعداد بیت ها را برای پیمایش زیاد کنیم، زمان جستجو کاهش پیدا می‌کنه اما ما باید جدول را تبدیل کنیم به یک درخت multi-bit trie. => همه‌ی prefix هایی که در جدول داریم، باید تبدیل کنیم به بیت های با ضرب 3. مثلاً 1* را باید تبدیل کنیم به 100* ، 101* ، 110* ، 111* اینا P_2 هستن و می در جدول نگاه می‌کنیم مثلاً 101* ، P4 است، می‌ذاریم P4.

=> در جدول اول، $P_2 \leftarrow 100$ ، $P_4 \leftarrow 101$ ، $P_2 \leftarrow 110$ ، $P_5 \leftarrow 111$

P1 که * بود، می‌شه 000* تا 111* . اون چهارتا که با 1 شروع می‌شده دیگه P1 نیست. چون 00* بوده P3، پس 000 ، 001 می‌شه P3. ...

maxnote

Scanned with CamScanner

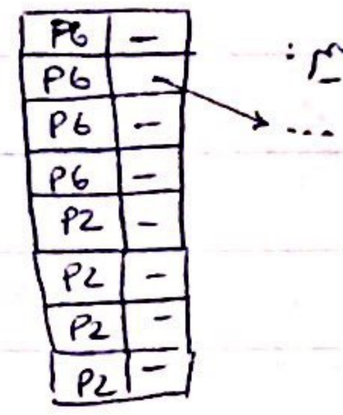
⇐ در این روش، تعداد بیت های prefix در جدول forwarding را می‌کنیم مضربی از 3، اون prefix هایی که طول بزرگتری دارند (مثل P4 نسبت به P1)، اینو جایگزین می‌کنیم تا جدول درست میشه.

- نگاه می‌کنیم می‌بینیم ما مجبور شدیم P3 را دوبار تکرار کنیم، P1 را 2 بار و ... ← تعداد تکرارهای زیادی داریم.

- در هر ساختار داده، 2^k ، pointer را باید نگاه داریم. برای یه دونه، $\frac{W}{K}$ از این جداول باید داشته باشیم ضرب در 2^k ، تعداد entry ها هم N تا بود ⇐

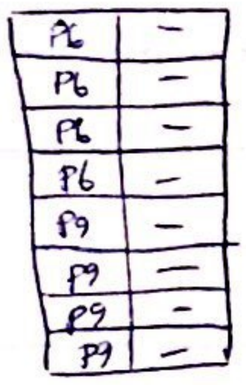
مرتبه‌ی حافظه‌ی مصرفی : $O(\frac{W}{K} \cdot 2^k \cdot N)$

p.31 : اگر روش disjoint prefix را اجرا کنیم که می‌آدیم pointer ها را push می‌کردیم به سمت برگ ها، مثلاً در جدول ص 30 ، P2 یه pointer داره. اگه P2 را حذف کنیم، جاش پونیتر P1 را بگذاریم و P2 ها را به این صورت قرار دهیم:



، اینو الان push کردیم به سمت برگ ها.

به همین ترتیب، P6 را push کنیم به نود 3 :



نهایتاً ساختار ص 31 به وجود میاد. در جدول هر نود، یا next-hop را نگه می‌داریم یا pointer را.

⇐ تقریبا حجم حافظه نصف می‌شه. چون در جدول قبلی هم prefix داشتیم هم پونیتر، این جا یا Ptr داریم یا prefix. ⇐ حجم حافظه کم می‌شه ولی از نظر order فرقی نمی‌کنه (به تقسیم بر 2 می‌شه.)

p.32 : کارایی : زمان جستجو می‌شه $O(W/K)$. حجم حافظه‌ی مصرفی : $O((2^k \cdot N \cdot W)/K)$

نکته ای که هست اینه که اگه بخوایم در جدول forwarding ، یک گره را باید تغییر دهیم، یک entry اضافه کنیم مثلاً، باید پیمایش کنیم گره را پیدا کنیم که می‌شه زمان $\frac{W}{K}$ ، بعد در بدترین حالت مجبور می‌شویم که اون k تا ptr یا next-hop ، گره را تغییر دهیم ← 2^k ⇐ $O(\frac{W}{K} + 2^k)$

مثلاً در ص 30 ، اگه P9 را تغییر دهیم، $\frac{W}{K}$ طول می‌کشه تا برسیم به اون گره. بعد بردیم در Node3 ، در بدترین حالت، همه‌ی entry هاشو تغییر دهیم.

⇐ در روش multi-bit trie : ✓ زمان جستجو در مقابل binary trie بهبود پیدا می‌کنه. ($O(W) \to O(W/K)$)

✗ حجم حافظه‌ی مصرفی افزایش پیدا می‌کنه که access time افزایش پیدا می‌کنه.

افزایش حافظه مصرفی ⇐ $\frac{2^k}{K} > 1$ ← $O((2^k \cdot N \cdot W)/K)$ ← $O(NW)$

✗ زمان به روز رسانی افزایش : $O(W) \to O(W/K + 2^k)$

Scanned with CamScanner

مرور : روش های جستجو در جدول forwarding را دیدیم که دو دسته بود :

1) نرم افزاری ← معمولاً مبتنی بر درخت trie

2) سخت افزاری

p.30 : روش multi-bit trie :

دقت multi-bit می‌کنیم، عمق درخت کاهش پیدا می‌کند. ⇐ تعداد دفعات مراجعه به حافظه کاهش ⇐ سرعت جستجو افزایش.

⇐ باعث می‌شه پیچیدگی حافظه‌ی مصرفی افزایش پیدا کنه. هر گره‌ی درخت باید 2^k اطلاعات prefix یا ptr در خودش نگهداری کنه ⇐ حجم حافظه‌ی مصرفی : $O((2^k \times N \times W)/k)$

⇐ و زمان به روزرسانی تغییرات : $O((2^k \cdot W)/k)$

- درخت multi-bit trie را با این روش های path compression , disjoint prefix می‌توانیم بهبود دهیم.

p.33 - تکنیک level-compression trie میاد از روی این جدول forwarding دو مرحله عمل فشرده سازی را روی درخت انجام می‌ده.

1) یه مرحله روی path compression انجام می‌ده مثل همون قبل.

2) مرحله‌ی دوم میاد multi-bit trie را استفاده می‌کنه که بهش می‌گه level compression

- در path compression، این شاخه های طولانی را میاد فشرده سازی روش انجام می‌ده، skip می‌کنیم. از روی بیت‌ها و skip number , skip value را نشون می‌دهیم. (مثل شاخه با برگ 2 در شکل)

- در level compression، هر جا درخت کامل شد، می‌آییم با multi-bit trie فشردش می‌کنیم. قبلاً در شکل (در درخت کامل)، 8 تا نود داریم، با LC همه را فشرده می‌کنیم. sride به جای 1 بیت، می‌شه 3 بیت. (روی این درخت شاخه های 9، 10، 11 که کامله هم فشرده سازی کرده)

- به این روش، LC می‌گویند که در دو مرحله از binary trie بهش می‌رسیم.

⇐ دو نوع فشرده سازی کردیم ⇐ حجم ساختار داده کاهش پیدا می‌کنه

کاهش دفعات مراجعه به حافظه (چون عمق درخت کم شده)، سرعت جستجو زیاد می‌شه.)

⇐ از نظر تجربی به دست میاد.

- از نظر order، با multi-bit trie فرق نداره. ولی ضریب در LC کمتر است. (از نظر عملی سرعت جستجو بالا)

maxnote

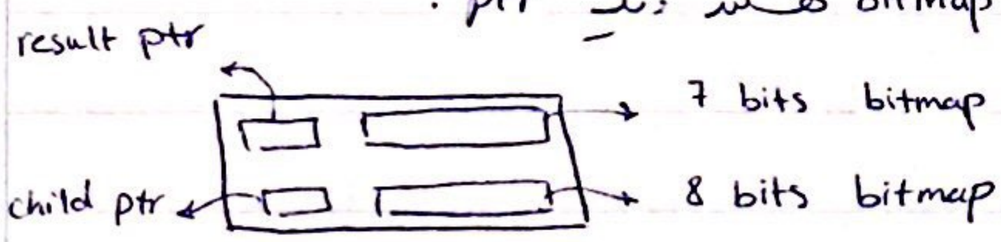
Scanned with CamScanner

- اشکال عمده‌ی روش : data structure ـ ثابت نداره چون بسته به نوع محتوای جدول forwarding ،
ممکنه یه دفعه در level ـ 3 ، فشرده سازی انجام بده ، یه دفعه در level ـ 2 .
مثلاً در اسلاید اگه کل اون درخت کامل در بخش b را می‌کنیم یک node ، ما باید 8 تا ptr نگهداری کنیم.
اما وقتی اون درخت کوچکتره در شکل را فشرده می‌کنیم در یک node ، باید 4 تا ptr (ریشه در شکل) این جا نگهداریم.
=> ساختار داده‌ی متغیر پیدا می‌کنه.
← پیچیدگی که این ساختار داده‌ی متغیر ایجاد می‌کنه ، اینه که نمی‌توانیم از یک data structure ـ ثابت استفاده کنیم و
پیچیدگی پیاده سازی را افزایش می‌ده . ← در صد مقاله باحث مونده و پیاده نشده .

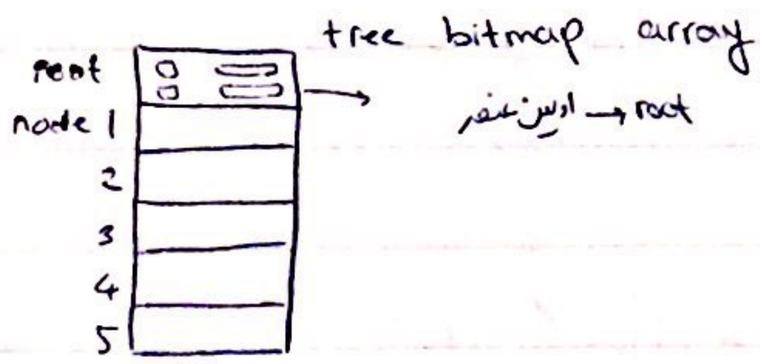
p.35 : Tree Bitmap Algorithm (الگوریتم درخت نگاشت bit)
← سعی کرده از ساختار داده‌ی ثابت استفاده کنه . ← مزیت
← سعی کرده با تکنیک های فشرده سازی ، حجم داده را به شدت کاهش دهد .

- اساس کار : میاد روی یک Stride ـ ثابت مثلاً 3 ، level compression انجام می‌ده.
وقتی درخت binary trie را داشته باشیم که از روی جدول forwarding ساخته شده ، می‌آییم از ریشه شروع
می‌کنیم ، در level ـ 3 ، اینها را جدا می‌کنیم. این می‌شه ریشه‌ی درخت bit map الگوریتم درخت نگاشت بیت.
دوباره بعد از level 3 ، جدا می‌کنیم. با این تقسیم بندی روی درخت binary ، node های درخت bitmap را
مشخص می‌کنیم. ← در شکل اسلاید ، شده نود ریشه و نود 6 تا 5 .

- هرکدوم از این node ها یک ساختار ثابت دارند که شامل 2 تا bitmap هستند و یک ptr .
=> هرکدام از نودها ، 4 فیلد اطلاعاتی دارند :



- 2 تا جدول داریم. یک جدول که این نودها را درخودش نگه می‌دارد. مثلاً در این مثال 5 تا نود داشتیم که هرکدام، یک خانه‌ی
جدول هستند :

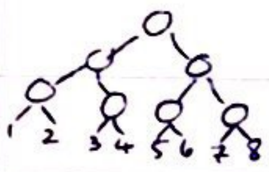


- یک جدول دیگه داریم که اطلاعات next hop را توش ذخیره می‌کنیم :

Scanned with CamScanner

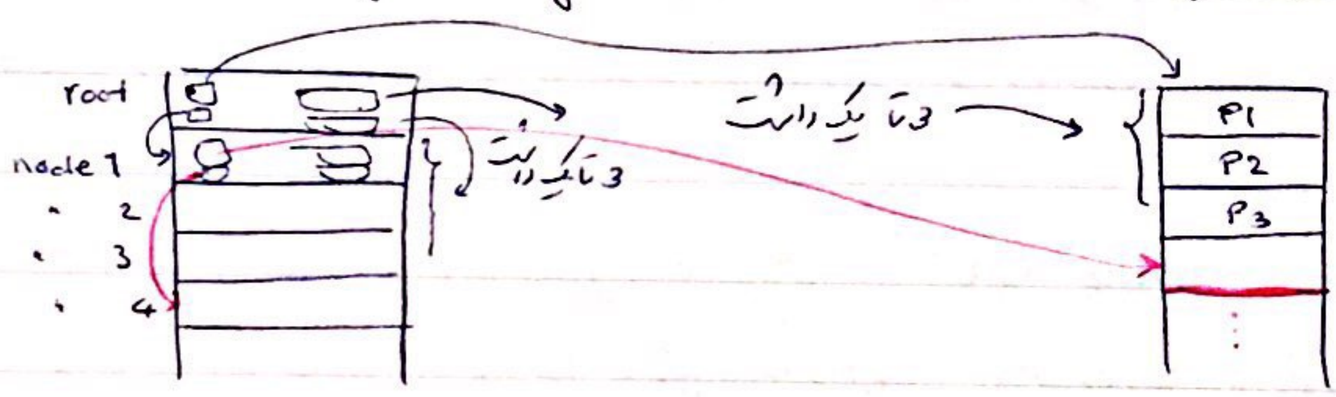
اسلاید ص 35 : اون جا که خط چین گذاشته، نود root است. 7 تا bitmap داره. اگه دقت کنیم در هر نود حداکثر می‌تونه 7 تا next-hop داشته باشه. ترتیب bitmap را هم نشون داد:
این می‌شه 7 تا بیت. هر جا اون نود، اطلاعات next hop دارد (خاکستری است در شکل)، یک می‌گذاریم
=> 1011 000

- یه result ptr داریم که اشاره می‌کنه به جدول next hop array یا آرایه‌ی result، به اولین 1 که در bitmap هست. => P1 ، 1 بعدی، می‌شه خونه‌ی بعدی و ...
=> اگه بخواهیم بدونیم next hop یه نود کجاست، ptr را می‌گیریم بعد می‌گیم قبلش چند تا یک داشتم؟ مثلاً یه دونه. یکی به ptr اضافه می‌کنیم و به جدول مراجعه می‌کنیم.



- هر نود این درخت tree bitmap، حداکثر می‌تونه 8 تا ptr داشته باشه به نودهای دیگه:
اون جاهایی که ptr وجود ندارد، 0 می‌گذاریم، جاهایی که وجود دارد، 1 می‌گذاریم.
مثلاً برای شکل اسلاید: 0000 1101

- child ptr اشاره می‌کنه به اولین نود که نود 1 است و بعدی و بعدی...
=> tree bitmap array next hop array



=> درخت به این شکل درست می‌شه ← P. 36 (2 تا آرایه‌ی result، tree bitmap array را کشید.)

- چه حسنی دارد؟ Structure ثابت. 2 تا آرایه‌ی ثابت داره، حجم حافظه‌ی مصرفی نسبتاً کم است
چون به ازای هر 8 تا نود، ما دو تا ptr داریم و یک bitmap 7 بیتی و یک bitmap 8 بیتی. اطلاعات next hop را هم که ذخیره کردیم.

- این روش در روتر Cisco CRS6 استفاده شده.

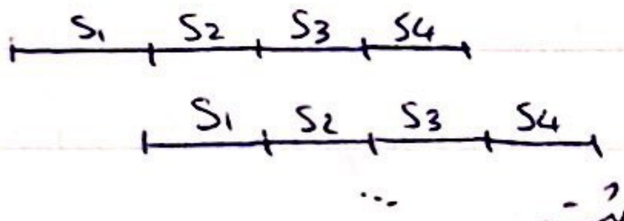
- مرتبه‌ی حجمو: $O(W/k)$
حجم حافظه‌ی مصرفی به ازای هر نود: دو تا bitmap بود یکی 2^k یکی 2^k-1 => مرتبه‌ی 2^k => $O(2^k \cdot N \times W/k)$
که از نظر مرتبه مثل multibit trie ولی این جا 2^k بیت است در multibit trie، 2^k ptr است. => ضریب این خیلی کمه.

maxnote

Scanned with CamScanner

p.38 : می‌دانیم اگر یک کارهای sequential داشته باشیم که طولانی انجام می‌شوند، ما می‌توانیم با pipelining سرعت را افزایش دهیم.

- pipelining : خط تولید است. اگر مثلاً یه کاری 4 Stage ، طول می‌کشه انجام بشه، می‌تونیم این Stage ها را از هم مستقل کنیم وقتی مثلاً کار در S1 تموم شد، بدیم به S2 ، در S1 یه کار جدید را شروع کنیم و این کار را ادامه دهیم :



=> وقتی pipeline پر می‌شه، ما در هر timeslot ، یک خروجی خواهیم داشت.
این کاری است که در خط تولید انجام می‌شه.

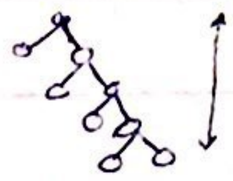
- آیا می‌شه این زمان جستجو که نسبتاً طولانی است با تکنیک pipelining ، کاهش دهیم؟ بله
کافیه درخت را در 4 عمق بشکنیم ، عمق اول را بدهیم به پردازنده‌ی 1. نتیجه‌ی اینو بدیم به پردازنده‌ای که در عمق دوم کار می‌کند و... و نهایتاً result به پردازنده‌ی عمق چهارم می‌رسه.

=> وقتی پردازنده‌ی اول جستجو را تموم کرد و داد به پردازنده‌ی دوم ، می‌تونه یه جستجوی جدید را شروع کنه...
=> وقتی pipeline پر می‌شه، معادل زمان هر پردازنده (معادل هر مرحله)، ما یک خروجی خواهیم داشت.

p.39 : پس 4 تا پردازنده داریم، عمق 1 (S1) را می‌دهیم به پردازنده‌ی 1 ، عمق 2 را به پردازنده‌ی 2 ، ... چون اینجا حافظه‌هاشون از هم مجزا است ، بنابراین ، پردازنده‌ی 1 که کارش را تموم می‌کند ، نتیجه را داد به پردازنده‌ی 2 ، ادامه‌ی کار را پردازنده‌ی 2 در پیمایش انجام می‌ده => پردازنده‌ی 1 می‌تونه یک جستجوی جدید را شروع کنه.
=> باعث می‌شه سرعت جستجو 4 برابر شود.

p.40 : روش Binary Search on Prefix Range

- می‌دانیم binary search سریعترین روش سرچ است که می‌شه استفاده کرد.
- چرا binary search روش خوبی است؟ چون درخت جستجو ممکنه نامتوازن باشه. مثلاً باید کل عمق را جستجو کنیم برسیم به گره‌ی آخر.



در بازیابی سرچ چون می‌آییم داده‌ها را 2 بخش می‌کنیم ، از وسط شروع می‌کنیم هی تقسیم می‌کنیم ، یه جوراییی داریم درخت متوازن ایجاد می‌کنیم. => عمق سرچ کمتر از حالت عادی می‌شود.

- در binary search می‌آییم داده‌ها را Sort می‌کنیم. مثلاً : 45 30 27 | 22 15 10 داده‌ها را 2 بخش می‌کنیم. عددی که می‌خواهیم جستجو کنیم (مثلاً 30) را مقایسه می‌کنیم با عدد وسطی. چون 30>27 ، می‌رویم سمت راست...

Scanned with CamScanner

- آیا binary search را می‌توان روی جدول forwarding اجرا کرد؟

به شکل اینکه entry های جدول یک عدد نیستند بلکه یک رنج هستند. مثلاً 1* یعنی 100...0 تا 11...1

اگر می‌خواهیم binary search استفاده کنیم باید از binary search on Prefix range استفاده کنیم.

برای این کار یکی از ایده‌هایی که می‌شد استفاده کرد اینکه بیایم هر entry را با آدرس ابتدایی و انتهایی‌اش نشان دهیم. مثلاً P1: 100...0 - 11...1 ... ← شکل c در اسلاید. 3 تا رنج شد 6 تا عدد.

حالا اگر جستجو کردیم داخل یک رنج قرار گرفتیم، مثلاً نتیجه در داخل رنج P1 قرار گرفت، آیا سته برای شبکه با next hop P1 است؟ شبکه دارد. ممکنه داخل رنج P1، یک رنج P2 باشه یا نه مثلاً رنجی وجود نداشته باشه.

اگه دقیقاً مساوی اون عددها شه که متعلق به خودشه، اگه داخلش قرار بگیره، یعنی بزرگتر از شروع رنج باشه، می‌آییم مشخص می‌کنیم. مثلاً

100000 P1
← P1
101000 P2
← P2
101111
⋮

⇐ اولا رنج‌ها را به 2 تا آدرس ابتدا و انتها تبدیل می‌کنیم. ثانیاً برای هر entry، 2 تا next hop باید نگهداری کنیم. یکی حالت مساوی، یکی در حالتی که بزرگتر از آدرس است. و این حالت بزرگتر بودن، قاعده‌ای نداره. باید اون شکل (c) را درست کنیم، پردازش کنیم.

← مرتبه‌ی جستجو: 2N entry داریم، چون binary search است، تقسیم بر 2 می‌شه، ⇐ مرتبه: $O(\log_2 N)$

اگر چندبیت چندبیت اینو بشکنیم، مثلاً جدول را 4 قسمت کنیم، دو بیت اول را نگاه می‌کنیم در کدوم بخش هست، یعنی به جای 2 بیت، q بیت را در نظر بگیریم، مرتبه‌ی سرچ می‌شه: $O(\log_q N)$

p.41: ایراد این روش، پیچیدگی update است. برای هر تغییر باید اطلاعات nexthop را به روز کنیم. ← $O(N)$

حافظه‌ی مصرفی 2N بود که می‌شه مرتبه‌ی: $O(N)$

p.42: روش‌های مبتنی بر سخت‌افزار

- دیگه ما برنامه‌ی جستجو نداریم. یک مدار سخت‌افزاری داریم آدرس مقصد را بهش می‌دیم، اون به ما next hop را با یه تأخیری می‌دهد: (مثلاً یکی دوبار سه‌بار مراجعه به حافظه داره)

dst addr. → [مدار سخت‌افزاری] → next hop
تأخیر

- اشکال عمده‌ی روش سخت‌افزاری: انعطاف‌پذیری کم. یعنی اگر حجم جدول سریع‌تر افزایش پیدا کنه، با توجه به اینکه مدار را برای یه حداکثر حجمی طراحی کردیم، ممکنه scalability نداشته باشه.

اشکال دوم: به روزرسانی ممکنه پیچیده باشه (البته همه‌ی روش‌ها، همه‌ی این مشکلات را باهم ندارن)

اشکال سوم: انرژی مصرفی بالا

اشکال چهارم: هزینه‌ی پیاده‌سازی بالا

maxnote

Scanned with CamScanner

- حسن روش : سرعت جستجوی بالا.

p.43 : روش های سخت افزاری به طور کلی به 2 گروه دسته بندی می شه :

1) روش مستقیم یا direct (2 تای اول)

2) روش های قابل آدرس دهی براساس محتوا یا Content addressable mem. (CAM) (4 تای دوم)

- از آنجا که در حافظه های CAM، آدرس را مثلاً به صورت host ID ← محل عبور پاکت * نشان می دهیم، اما در عمل باید بتوانیم در این حافظه ی CAM، سه مقدار 1 و 0 و don't care را ذخیره کنیم.

Prefix

=> از حافظه های ternary CAM یا TCAM استفاده می کنیم. T یعنی می توانیم در هر خانه ی این حافظه، 1، 0 یا don't care را ذخیره کنیم.

- در عمل چطوری این رو پیاده می کنیم؟ در عمل 2 تا در نظر می گیریم : 100 ... 0 (ذخیره) ، 110 ... 0 <= 10xxx...x (don't care)

می گیم کدوم ها don't care هستند (0 ها)

=> یه مقدار هزینه ی پیاده سازی حافظه افزایش پیدا می کنه چون می خواهیم 3 تا مقدار را نشون بدیم مجبوریم به ازای هر بیت، 2 بیت حافظه مصرف کنیم. یکی بگیم اصلاً don't care هست یا نه، دوم اگه don't care نیست، مقدارش چیه.

- در روش direct این شکلی عمل می کنیم. جدول forwarding را در حافظه می گذاریم، آدرس مقصد می شه آدرس حافظه. یعنی مستقیماً با آدرس مقصد، حافظه را مقداردهی می کنیم و داخلش اطلاعات next hop را ذخیره می کنیم.

آدرس مقصد → [ ] → next hop

=> با یکبار مراجعه به حافظه، next hop به دست میاد.

- ایراد روش direct : آدرس، یک عدد 32 بیتی است. => یک حافظه می خواهیم به طول 2^{32} یعنی 4G. حالا اگه هر خونه ی حافظه هم یه عدد 2 بایتی باشه، می شه 8GB حافظه به ازای هر جدول forwarding.

اگه فرض کنیم روتر ما 100 پورت ورودی/خروجی داشته باشه، به ازای هر پورت که forwarding engine در پورت ها پیاده شده، 8GB حافظه مصرف می شه برای جدول forwarding. ← این 1 مسأله که حجم حافظه خیلی زیاده.

↳ 800 GB حافظه ی RAM

مسأله ی دوم : به روزرسانی : چون هر entry در جدول forwarding یک رنج است، پس باید اون رنج را برای همه ی next hop ها یکسان بذاریم. => تکرار در حافظه زیاد داریم ← حجم حافظه زیاد و مشکل دوم هم بحث به روزرسانی است. بخواهیم به روزرسانی کنیم، مجبوریم تعداد زیادی خانه ی حافظه را تغییر دهیم.

Scanned with CamScanner

روش هایی مثل DIR-24-8 ، BC-16-16 . روش های direct هستند که سعی کردن هم مجم حافظه‌ی مصرفی را کاهش دهند ، هم به روزرسانی را ساده‌تر کنند .

DIR-24-8

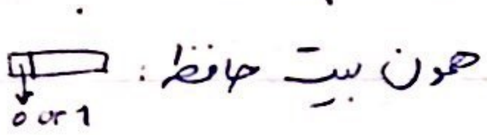
p.44 : دقت توزیع prefix ها در جدول forwarding را دیدیم ، بیشتر prefix ها در محدوده‌ی 16 تا 24 بودند .

- روش 24-8 می‌گه ما درختی که داریم را در level = 24 می‌شکنیم . چه قدر حافظه باید این جا مصرف کنیم ؟ 2^{24} یعنی 16M .

بعضی از prefix ها در جدول forwarding طول بزرگتر از 24 دارند . (تعداد کمی) . برای اونا یه حافظه‌ی سطح دوم می‌گذاریم . هرکدوم از این 24 تا 32 یعنی 8 بیت ، $2^8 = 256$. ← تعداد کمی از اینا داریم .

=> یه حافظه‌ی سطح 2 می‌گیریم برای prefix هایی که طول بزرگتر از 24 دارند ، به جای اینکه next hop را ذخیره کنیم ، یه ptr می‌زنیم به حافظه‌ی دوم و اطلاعات next-hop در حافظه‌ی دوم است .

p.45 : یه ورودی داریم که 32 بیت آدرس مقصدی است که از روی pkt برداشته‌ایم . اول 24 بیت اول را بررسی می‌کنیم مراجعه می‌کنیم به حافظه‌ی اول . 2 حالت داره ، یا next-hop را پیدا می‌کنیم یا خیر . یک بیت در حافظه است که اگر next hop هست ، اون بیت ، 0 است ، اگر ptr است به حافظه‌ی دوم ، 1 است .

=> اگه صفر باشه که اطلاعات next hop را داریم . اگه بیت 1 باشه ، یه ptr است به حافظه‌ی دوم که به ازای هر کدوم از این رنج ها 256 خونه داریم . ← ابتدای جدول . بقیش ، آدرس دهی داخلی می‌شه ، next hop به دست میاد و از این  به خروجی می‌رسه . Select ← multiplexer وصله به همون بیت حافظه :

=> 2 تا حافظه داریم : 2^{24} ← 16M که اگه next-hop یا ptr را با 2 بایت نشون بدیم ← 32 MB
حافظه‌ی دوم ، چون prefix ها حیلی کمه ، می‌شه مثلاً 1MB بگیریم . => جمع : 33 MB

p.47 : مثال : فرض کنیم یه جدول forwarding داریم با 3 تا entry ← A , B , C
جدول سطح یک را می‌بینیم ، چون 24 بیت اول را استفاده می‌کنیم ، A که 16 بیت داره ، 256 تا خونه‌ی A در جدول داریم . اونایی که استثنا هستند ، ptr می‌خورن به جدول دوم .
در بدترین حالت ، اگه یه prefix = 1 بیت ، تغییر کنه ، 2^{23} خانه‌ی حافظه‌ی سطح اول باید به روز شود . مثلاً اگه default gateway که prefix = 0 بیت است تغییر کنه ، 2^{24} خانه‌ی حافظه باید به روز شود . => زمان به روز رسانی حافظه‌ی اول در بدترین حالت : 2^{24} زمان به روز رسانی حافظه‌ی دوم : 2^8

- حافظه‌ی مصرفی زیادتر از اندازه‌ی جدول است به خاطر تکرار هایی که داریم .

maxnote

Scanned with CamScanner

P.48 : با یک حافظه‌ی dynamic Ram با تأخیر 50 ns ، می‌تونیم 20 میلیون lookup در ثانیه انجام دهیم. ← performance با

- معایب : 1) استفاده‌ی زیاد حافظه 2) تغییرات زیاد برای یک به روزرسانی

P.49 : روش Bitmap Compression

شبیه به DIR 24-8 است با این تفاوت که چند بهبود بهش اضافه کرده :

1) اومده اون level ـ 24 را level ـ 16 گرفته. ⇐ حافظه سطح اول : 2^{16} (64K) حافظه‌ی سطح دوم هم به ازای هر prefix که طول بزرگتر از 16 داشته باشه : 2^{16} ← بزرگ شده

این روش اومده این 2^{16} را تغییر کرده. گفته ما چک کنیم ببینیم چه قدره. اگه مثلاً k است، اینو می‌کنیم 2^k. بدترین حالتش 2^{16} است.

(مثلاً ما یه درخت باینری داریم که تا سطح 16 شکستیم، حالا این گاهی ادامه پیدا می‌کنه تا سطح 24 ، 25 ، ... این 16 تا هرچی بود را k در نظر می‌گیریم.)

⇐ برای سطح دوم، حافظه‌ی fix ـ 2^{16} را نمی‌گیریم، بستگی به عمق بعد از 16 در نظر می‌گیریم.

2) در روش DIR 24-8 ، تکرار ما زیاد داشتیم. بیاد با روش bitmap ، اینا رو فشرده می‌کنه که حجم حافظه را کاهش دهد.

P.51 : وقتی سطح را در 16 شکست، خانه‌ی سطح یک، یک بیت داره، که مشخص می‌کنه اگه :

next hop ← 0
ptr ← 1

یک k هم نگه می‌داره که می‌گه چند تا دیگه ادامه پیدا می‌کنه. k می‌تونه از 0 تا 15 باشه. ← 4 بیت یعنی تا عمق 16.

در حافظه‌ی سطح دوم، برای این prefix ، چند تا خونه نگهداری می‌کنه؟ k تا.

⇐ از اون 32 بیت آدرس مقصد که از هدر بسته برداشتیم، 16 بیت اول را استفاده می‌کنیم برای آدرس دادن سطح یک.

این یه ptr به ما می‌ده به حافظه‌ی سطح دو. از این مابقی 16 تا، چند تا رو باید استفاده کنیم؟ k

← k تا را برمی‌داریم، می‌آییم در حافظه‌ی سطح 2 ، next hop را به دست می‌آوریم.

- منتهی حافظه‌ی سطح دوم را هم میاد فشرده می‌کنه:

Scanned with CamScanner

P.52: میاد می‌گه حافظه‌ی next hop array اگه حافظه‌ی سطح دوم باشه که تکرار توش زیاده، میاد یه bitmap درست می‌کنه که به ازای اولی، 1 می‌گذارد، بعدی‌ها که تکرار هست، صفر می‌گذارد. اولین تغییری که می‌بینید 1 می‌گذارد، بقیه را صفر می‌گذارد.

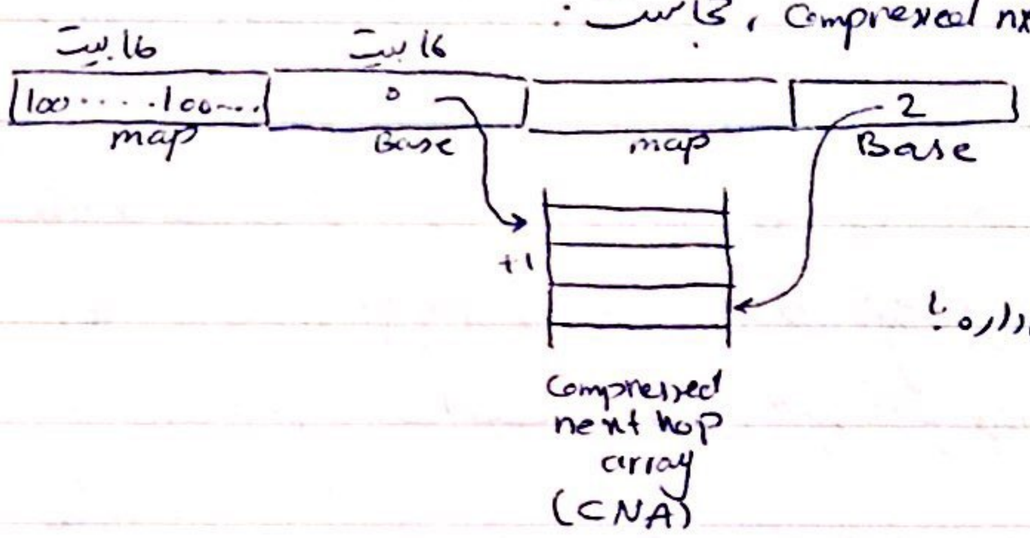
- این عددا (مثل 2, 8, 7, 6, ...) را میاد در حافظه می‌گذارد، اضافه‌ها را حذف می‌کنه. یه حافظه‌ی compressed next hop array به دست میاد که تکرار توش وجود ندارد.

به جای اون 22222 82 .... که در سطح دوم ذخیره کنه، میاد حافظه‌ی فشرده شده و اون 10001... (bitmap) را ذخیره می‌کنه که حجمش به مراتب کمتر از قبلیه.

- این 2 تا را چه طوری ذخیره می‌کنه؟ هر 16 بیت، bitmap را می‌گیرد. یه ptr می‌زنه به حافظه‌ی سطح دوم که مربوط به اولین یک است. (compressed next hop array)

P.53: اولین یک، آدرسش اینه که next hop توی compressed nxt hop array، کجاست:

دومین یک، اون base، به علاوه‌ی 1 می‌شه...



=> به جای اون 2 تا جدول، میاد Codeword array را نگه می‌داره با اون CNA.

P.54: مدار سخت‌افزاری که به دست میاد به این صورته. ← اسلایدِ. 32 بیت آدرس مقصد را برمی‌داره، 16 بیت اول را استفاده می‌کنیم برای آدرس دهی حافظه‌ی اول. یا این ط next hop را به دست می‌آوریم یا نه و باید بریم حافظه‌ی سطح دوم.

می‌آییم حافظه‌ی سطح دوم. اگر k کوچکتر از 3 باشه، ارزش فشرده سازی نداره. => برای اونایی که k، از 0 تا 3 هستند، فشرده سازی نمی‌کنیم. از این 4 بیت استفاده می‌کنیم برای آدرس دهی. (اگر بیت‌های آدرس را از یک شماره گذاری کنیم (نه صفر)، بیت 1 تا 16 می‌شه بخش اول، بیت 17، 18، 19، 20 استفاده می‌شه برای آدرس دهی حافظه‌ی دوم.) ← next hop را به دست می‌آوریم بدون فشرده سازی.

اگر $k > 3$: مراجعه می‌کنیم به bitmap. چون هر 16 تا رو گذاشته بودیم در یک خانه‌ی حافظه. بنابراین، اون بخش دوم آدرس را (بعد از 16 بیت)، k هر عددی باشه، 4 بیتش را حذف کنیم، مابقی را استفاده کنیم برای آدرس دهی حافظه‌ی دوم با base ptr که از حافظه‌ی اول گرفته‌ایم. اون کلمه پیدا می‌شه، اون 4 بیت را مشخص کنیم کدوم بیت است؟ ← اون بیت را (در map) نگاه می‌کنیم، اگر صفر است، می‌رویم به سمت چپ، اولین یک را پیدا می‌کنیم، اگر یک بود، خودش.

نگاه می‌کنیم قبلش چند تا یک دیگه وجود دارد؟ مثلاً اگه یک عدد یک دیگه بود، آدرس را به علاوه‌ی یک می‌کنیم. (به علاوه‌ی تعداد یک‌ها می‌کنیم)، compressed next hop array را آدرس می‌دهیم، next hop را به دست می‌آوریم.

maxnote

Scanned with CamScanner

=> مثل روش direct است با این تفاوت که در روش DIR 24-8 ، یا از حافظه‌ی اول به دست می‌آید یا با مالتی بیت ها ، حافظه‌ی سطح دوم . این جا می‌گه از مالتی بیت ها ، k بیت را انتخاب کن ، اگه k<3 بود که مستقیم به حافظه‌ی دوم مراجعه کن . اگر k>3 ، از اون k بیت ، 4 بیت را کنار می‌گذاریم ، اون base ptr (که از حافظه اول در وبد) با bit های آدرس ، آدرس حافظه‌ی دوم را به ما می‌دهد . از اون 4 بیت که کنار گذاشته بودیم ، از 16 بیت درون bitmap ، بیت مورد نظر را پیدا می‌کنیم ، اولین یک نزدیک به این را پیدا می‌کنیم ، نگاه می‌کنیم در اون کلمه ، چندمین 1 است ؟ به تعداد یک های قبلش به آدرس اضافه می‌کنیم (به base ptr) و اطلاعات next hop را به دست می‌آوریم .

p.52 : در واقع به جای اینکه به ازای هر 1 درون bitmap ، یک ptr ذخیره کنیم به CNHA ، اومد bitmap را 16 بیت 16 بیت جدا کرد ، به ازای هر 16 بیت ، یک ptr ذخیره کرد و از روی اینکه چندمین یک در bitmap باشه می‌شه آدرس را پیدا کرد . (در واقع ptr اشاره می‌کنه به اولین یک ، برای یک بعدی ، به ptr یکی اضافه می‌کنیم .)

حالا وقتی می‌خواهیم خونه‌ی 16 بیتی bitmap را آدرس دهیم ، k بیت آدرس داشتیم که k-4 بیت آن ، داره map (ص 53) را آدرس می‌دهد ، 4 بیت کم ارزش داره شماره بیت را مشخص می‌کنه . مثلاً بیت دهم ، یک است ، قبلش چند تا یک بوده ؟ یکی => به ptr یکی اضافه کن ، next hop به دست می‌آید .

p.55 : روش bitmap compression را پیاده سازی کردند برای یک جدول میلیاردی با 40000 entry ، 470 kB حافظه مصرف کردن . (زیر 1MB . در روش Direct ، 4G حافظه می‌خواست ، Dir-24-8 ، 33 MB حافظه می‌خواست .)

با یک حافظه‌ی SRAM => با access time => 10ns می‌تونه 100 میلیون lookup در ثانیه انجام دهد .

- اشکال روش : به روزرسانی . اگه یک آدرس تغییر کنه ، تمام جدول باید به روز شود . اگه اینو بشه به گونه‌ی دیگری روش کار کرد ، مثلاً 2 تا جدول داشته باشیم ، رو یکی update کنیم بعد در یک لحظه جداول را switch کنیم می‌شه ولی به هر حال به روزرسانی چالش داره در این روش .

- در TCAM هم حجم حافظه مصرف نمی‌شه هم هزینه‌ی به روزرسانی خوبه . در روش سخت افزاری ، TCAM مورد قبول است .

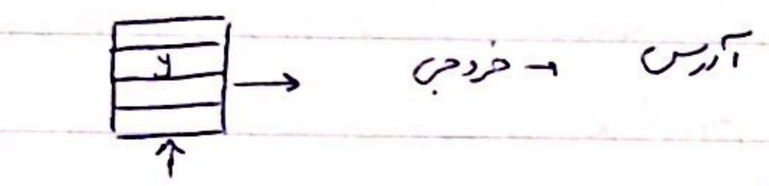
Scanned with CamScanner

مرور : روش های سخت افزاری جستجو را دیدیم که به 2 گروه تقسیم می شد : 1) روش های مستقیم
2) روش های مبتنی بر CAM (content addressable memory)

مجازی

- ویژگی روش های مستقیم : با یک یا دو بار مراجعه به حافظه، اطلاعات next hop به دست میاد برای pkt forwarding.
اشکال : ① حجم حافظه ی زیاد مصرفی . ② به روزرسانی جدول forwarding.

- در روش های مبتنی بر CAM :
در حافظه های CAM ، اطلاعات را در حافظه ذخیره می کنیم، بعد ورودی که بهش می دهیم محتوا هست ، خروجی که داده می شه آدرس هست که این محتوا آنجا ذخیره شده .



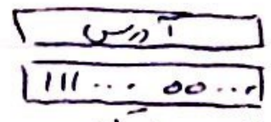
- اگه مثلاً عدد y را ورودی دهیم ، آدرس را می دهد. اگه y در چندین خانه ی حافظه ذخیره شده باشه ، معمولاً در CAM ، بین اینها یک اولویت قائل می شویم و آدرس یکی را می دهیم.

- در پیاده سازی این حافظه ها، به ازای هر خانه ی حافظه ، یک مدار مقایسه کننده وجود دارد که اون مقدار محتوا که مشخص می کنیم، با مقدار خانه ی حافظه مقایسه می شود و نتیجه اش 0 یا 1 هست.
از نظر پیاده سازی ، این حافظه ها گران قیمت هستند چون به ازای هر خانه ی حافظه یک مقایسه کننده وجود دارد .
اگه کلمات حافظه ، 32 بیتی باشند ، به ازای هر خانه ی حافظه ، یک مقایسه کننده ی 32 بیتی وجود دارد . اگه مثلاً حافظه 1k word ، خانه داشته باشه ، 1k هم مقایسه کننده لازمه .
در عوض، چون مقایسه به صورت موازی در همه ی خانه های حافظه انجام می شود، سرعت پیدا کردن خیلی بالاست .

- اگه این مقدار در چندین خانه باشه ، بین اینها یک اولویت می گذاریم ، اونی که بالاترین اولویت را دارد انتخاب می کنیم.
معمولاً اولویت را بر حسب آدرس می گذارند . ← اونی که آدرس کمتر دارد ، اولویت بیشتری دارد .

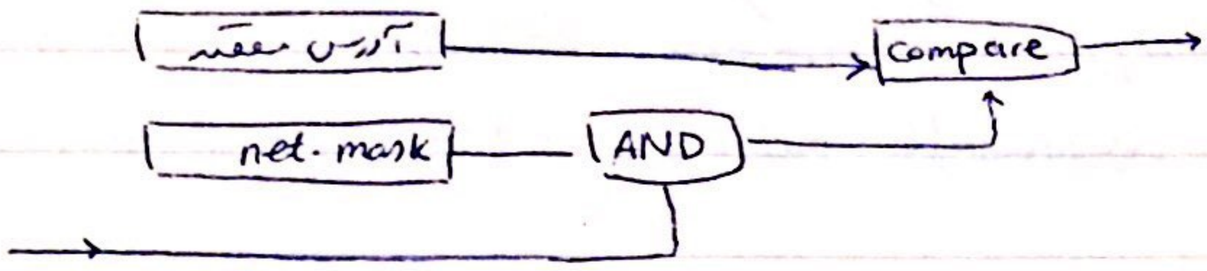
- ما وقتی در جدول forwarding می خواهیم جستجو کنیم ، اول دنبال آدرس شبکه ی مقصد می گردیم . وقتی آدرس را از هدر بسته برمی داریم ، وقتی می خواهیم مقایسه کنیم با جدول forwarding ، باید net. mask را در نظر بگیریم و به اندازه ی اون تعداد بیت را مقایسه کنیم. ⇐ ما باید 0 و 1 و don't care را در نظر بگیریم ⇐ حافظه ای می خواهیم که 0 و 1 و don't care را داشته باشند.

- در عمل به این صورت پیاده می شه که ما آدرس را قرار می دهیم ، network mask هم در یک خانه ی 32 بیتی نگه می داریم . (Net ID ← 1 ما بقی ← 0) :



وقتی می خواهیم یک آدرس را مقایسه کنیم، ابتدا آدرس را با net. mask ، and می کنیم ، بعد مقایسه :

maxnote
Scanned with CamScanner



=> به ازای هر خانه‌ی حافظه، net mask هم باید نگه داریم. ← 2 تا خونه‌ی 32 بیتی. و یک مقایسه‌کننده داشته باشیم.

← به این حافظه‌ها، "TCAM" گفته می‌شود.

=> جدول forwarding را در حافظه می‌گذاریم، وقتی آدرس می‌آید مقایسه می‌کنیم، چون net. mask ها متفاوته، ممکنه با چندین خونه‌ی حافظه match شود.

نسبت متعلق به شبکه‌ای است که maximum prefix را داشته باشه. اگر بیایم جدول forwarding را بر اساس تعداد بیت های net. mask مرتب کنیم و در حافظه قرار دهیم، اون که net. ID بیشتری دارد، می‌شه خروجی و priority encoder (در اسلاید)، خروجی 1 را می‌دهد.

- متناظر با این حافظه‌ی CAM، ما یک حافظه‌ی RAM داریم که متناظر با اون آدرس، اطلاعات next hop را قرار دادیم.

=> 2 بار مراجعه به حافظه. یکبار به TCAM مراجعه می‌کنیم آدرس حافظه را به دست می‌آوریم. با اون آدرس حافظه به RAM مراجعه می‌کنیم، next hop را به دست می‌آوریم.

- اگر بخواهیم یکی از سطرهای جدول را تغییر دهیم، یا سطری اضافه یا کم کنیم، فقط با یکبار مراجعه به حافظه این کار انجام می‌شود. چون براساس تعداد بیت های net. ID اینجا را sort کردیم، برای اینکه اگه خواستیم سطر جدیدی اضافه کنیم نخواهیم کل جدول را shift دهیم، می‌توانیم بین net ID های با طول مختلف، تعدادی خانه‌ی خالی در نظر بگیریم که برای اصلاح کردن جدول یا درج آدرس جدید، فضا داشته باشیم.

=> مرتبه‌ی جستجو : $O(1)$ ← 2 بار مراجعه به حافظه (به تعداد entry ها وابسته نیست)

مرتبه‌ی به روزرسانی : $O(1)$

حجم حافظه‌ی مصرفی : $O(n)$

← بهترین وضعیت. ولی اشکال حافظه های TCAM ؟

1) هزینه (2 برابر طول آدرس حافظه می‌خواهیم، مقایسه‌کننده 32 بیتی، and = 32 بیتی)

2) در روترهای core، تعداد entry های جدول حدود 2 میلیون است. به ازای هر مقایسه که زمانش در حد ms است، 2 میلیون مدار مقایسه‌کننده همزمان دارن عمل مقایسه را انجام می‌دهند. اگر هر مقایسه‌کننده، 0.001 mA جریان بکشد، تقریباً در زمان ms، 2A جریان می‌کشد. حالا اگه در روتر core در نظر

Scanned with CamScanner

بگیریم که 500 پورت ورودی / خروجی دارد، به ازای هر پورت انقدر جریان داره میکشه. یعنی حجم انرژی مصرفی به اندازه‌ای بالاست که استفاده از اینها امکان پذیر نیست.

- چاره؟ سعی کنیم با حفظ ویژگی های مثبت ($O(1)$ زمان جستجو ، $O(1)$ زمان به‌روزرسانی ، $O(n)$ حافظه مصرفی) ، کاری کنیم که انرژی مصرفی را کاهش دهیم.

- انرژی مصرفی از کجا ناشی می‌شه؟ از مقایسه کننده های همزمان. اگر می‌خواهیم مصرف انرژی را کاهش دهیم، نباید کل حافظه را فعال کنیم و آدرسی که می‌خواهیم پیدا کنیم را در کل حافظه دنبالش بگردیم. اگر کل حافظه را فعال نکنیم، چطور جواب درست را پیدا کنیم؟ پاسخ در ادامه...

p.57 : روش TCAM بسیار روش مناسبی از نظر مرتبه‌ی جستجو و به‌روزرسانی و حجم حافظه‌ی مصرفی است. ایراد: مصرف انرژی. به ازای هر ترانزیستور، 10 تا 15 وات توان مصرفی است که اگر در کل زمان به ازای همه‌ی پورت ها فعال باشه، انرژی مصرفی خیلی بالا می‌رود.

p.58 : ایده‌ی اولیه‌ای که مطرح شد: کاهش تعداد سطرهای جدول forwarding، سعی کنیم اندازه‌ی جدول را کاهش دهیم و درنتیجه، مصرف انرژی را کاهش دهیم.

2 تا تکنیک برای این فشرده سازی پیشنهاد شده:

1) prefix pruning

2) mask extention (توسعه دادن mask)

p.59 : در روش prefix pruning می‌گه ما یک درخت جستجو داشتیم، (در اسلاید: اگر فرض کنیم جدول forwarding اولیه را داریم) نگاه کنیم می‌بینیم اگه با P4، match شود، پورت خروجی S است. با P3 هم اگه match بشه، باز پورت خروجی S است. (هر چند P3، استثنای P4 است، اما تفاوتی در forwarding نداره)

← بیایم نگاه کنیم کدوم محدوده‌ی آدرس معتبر (مثل P3 که زیرمجموعه‌ی P4 است)، اگر next hop یکسان دارد، حذف P3 از جدول، هیچ تأثیری در pkt forwarding ما ندارد.

← به این میگن prefix pruning (هرس کردن) ← آدرس پیدا کنیم که زیرمجموعه‌ی یک آدرس دیگه باشه و next hop برابر داشته باشه. مثل: P4, P3 P7, P1 P6, P2

=> می‌رسیم به یک جدول فشرده شده با entry های کمتر.

- این روش هر چند جواب میده و جدول را کوچکتر می‌کنه اما حجم کاهش، قابل مقایسه از قبل نیست و نمی‌دانیم

maxnote

Scanned with CamScanner

چه قدر داریم انرژی را کاهش می‌دهیم. => از قبل مجبوریم انرژی را تدارک ببینیم که اگه کاهش پیدا کرد و انرژی لازم داشتیم، امکانش را داشته باشیم. ← روش عملیاتی نیست هر چند خوبه.

- روش mask extention

P.60 : قبلاً اینطوری بود که به تعداد بیت های Net ID، در mask، یک می‌گذاشتیم، بقیه هم صفر. در این روش، می‌گه بیایم نگاه کنیم مثلاً P1 و P2، تعداد بیت های Net ID برابر دارند. مقایسه که می‌کنیم می‌بینیم آدرس این 2 شبکه به هم است فقط در بیت چهارم با هم اختلاف دارند. پورت های خروجی هم برابرن.

این 2 را بیایم با هم ترکیب کنیم: 100 x 11 x x

=> با این روش می‌شه آدرس شبکه های مقصد را که تعداد بیت های یکسان net ID دارند و مقصدهای یکسان دارند و فقط در یک بیت با هم اختلاف دارند، با هم ترکیب کنیم به یک entry در جدول.

- این هم شبیه prefix pruning است که هر چند همیشه کاهش اندازه‌ی جدول انجام می‌شود ولی از قبل قابل محاسبه نیست که میزان کاهش اندازه‌ی جدول چه قدر است. => نمی‌توانیم بگوییم چه قدر کاهش مصرف انرژی خواهیم داشت.

- بریم سراغ روش هایی که میزان کاهش انرژی قابل محاسبه دارند:

CoolCAMs

P.62 : cool از این جهت که هر چی مصرف انرژی کمتر، میزان گرما کمتر.

ایده: می‌گه اون حافظه‌ی TCAM که جدول forwarding را توش ذخیره کردیم و به N block تقسیم کنیم. مثلاً به 8 تا بلاک تقسیم کنیم.

آدرس شبکه های مقصد را براساس یک ID در بلاک ها ذخیره کنیم. مثلاً اگه سطرهای جدول باشه:

1100 * *
11011 *
⋮

بیایم مثلاً 3 تا بیت 3, 4, 5 را در نظر بگیریم برای همه‌ی entry های جدول، با در نظر داشتن 3 تا بیت، 8 حالت مختلف خواهیم داشت. مثلاً تمام آدرس های مقصد که این 3 بیت آنها 000 است را در بلاک صفر می‌گذاریم، 001 را در بلاک یک ...

=> جدول forwarding را به N block تقسیم می‌کنیم که از روی آدرس مقصد بسته می‌شه بفهمیم در کدوم بلاک را جستجو کنیم. مثلاً اگه سه تا بیت بسته‌ی رسیده، 000 بود، فقط بلاک صفر را فعال می‌کنیم و دنبالش می‌گردیم. آدرس که پیدا شد، SRAM را آدرس دهی می‌کنیم و next hop به دست میاد.

=> یک index ـ ثابت درست می‌کنیم از روی آدرس ها.

مثلاً جدول forwarding، N تا سطر داره، می‌خواهیم 3 بیت انتخاب کنیم که جدول را به 8 قسمت مساوی تقسیم کنیم،

Scanned with CamScanner

اون 3 بیت را انتخاب می‌کنیم ← c_0 و c_1 و c_2 . (لزماً هم پشت هم نیستن)

بسته که میاد، آدرس مقصد را نگاه می‌کنیم، بلاکی که باید جستجو کنیم مشخص می‌شه. ⇐ برای هر آدرس مقصد، فقط یکی از این بلاک‌ها (این 3 بیت) فعال می‌شه. اگه جدول را به k بلاک تقسیم کرده باشیم، هر بار $\frac{1}{k}$ حافظه فعال می‌شه.

⇐ کاهش انرژی مصرفی به $\frac{1}{k}$

مشکل این روش:

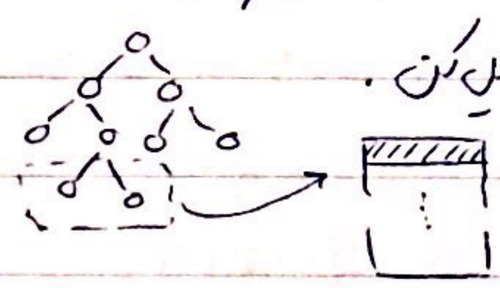
اون 3 بیت را از جدول forwarding به دست می‌آوریم. یعنی هر بار جدول forwarding را به روز کنیم، این 3 بیت را ممکنه تغییر دهیم. مشکل اینه اگه بخواهیم 8 قسمت کنیم، پیدا کردن 3 تا بیت کار سختی نیست ولی اگه مثلاً بخواهیم 5 بیت به بالا، کار ساده ای نیست. یعنی اگه بخواهیم جدول را به 8 قسمت تقریباً مساوی تقسیم کنیم، پیدا کردن 3 بیت، امکان پذیره ولی اگه بخواهیم به 100 قسمت تقسیم کنیم، تقریباً غیر ممکنه.

⇐ کاهش مصرف انرژی یه حدی دارد. مثلاً به $\frac{1}{8}$ یا $\frac{1}{16}$ می‌توانیم انرژی را کاهش دهیم. با توجه به اینکه انرژی مصرفی خیلی بالا بود، ممکنه این $\frac{1}{8}$ یا $\frac{1}{16}$ هنوز عدد مطلوبی نباشه.

راه حل؟ Trie-based table partitioning

p.63 : می‌خواهیم جدول را partition بندی کنیم، در روش قبلی، با bit selection این کار را کردیم که محدودیت داشت. روش قبلی بر درخت trie، می‌گه اندازه‌ی بلاک های حافظه را هر چی قدر می‌خواهی بذار ← b سطر در هر بلاک این b، ورودی الگوریتم است. هر چی می‌خواهی می‌تونی تنظیم کنی.

- جدول forwarding را به binary trie تبدیل کن.



- با یه روش پیمایش، اینو بشکن و بذار در هر بلاک به گونه‌ای که تعداد entry های جدول forwarding در این حداکثراً b تا باشه.
- برای هر تکه از درخت که جدا می‌کنی می‌ذاری در یک بلاک، یک index درست کن بذار در TCAM index.

⇐ یعنی یه حجم حافظه‌ی خیلی زیادی داریم، اومدیم اینو بلاک بندی کردیم، برای هر بلاک یه index درست کنیم. اگه مثلاً 1000 تا بلاک کردیم، 1000 تا هم index داریم.

- آدرس مقصد که میاد، اول index را پیدا می‌کنیم با یک abstract search (جستجوی سطح بالا). حجم حافظه‌ی index به مراتب کمتر از data TCAM است. index را که پیدا کردیم، آدرس مقصد را می‌ریم در اون بلاکی که index بش را پیدا کردیم، جستجو می‌کنیم و next hop را به دست می‌آوریم.

⇐ باید بتوانیم جدول forwarding را بر اساس روش trie تکه تکه کنیم، برای هر بلاک، index بسازیم بذاریم در index TCAM.

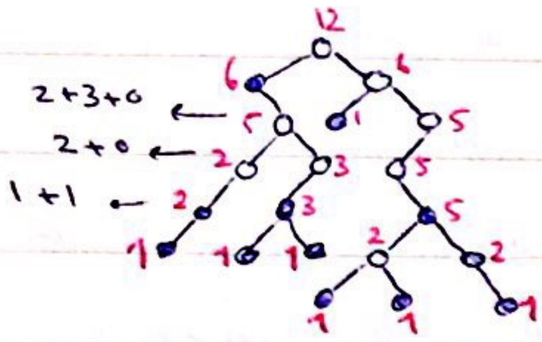
maxnote

Scanned with CamScanner

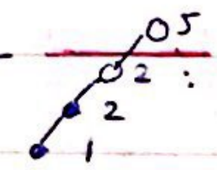
⇐ دستمون باز ه دیگه محدودیت نداریم حافظه را به چند بلاک تقسیم کنیم.

P.64 : در این روش که حافظه را تقسیم می‌کنیم به چند بلاک و برای هر بلاک یه index درست می‌کنیم، به این شکل عمل می‌شه که برای هر گره‌ی این شکل، یک value تعریف می‌کنیم به این ترتیب که: (گره های توپر، next hop را ذخیره کرده اند.)

- value ← برگ‌ها، یک است.
- value ← بقیه، می‌شه value ← فرزندانش به علاوه‌ی خودش. مثلاً



این value چی می‌گه؟ می‌گه اگه یه شاخه را بخواهی قطع کنی، چند خانه‌ی حافظه در TCAM باید براش نگه داری. مثلاً



← 2 خانه

پس value : تعداد entry هایی که در اون بلاک باید ذخیره کنی.

P.65 :
- در این روش، برای table partitioning، 2 روش ارائه شده:

1) Subtree-split Algorithm (الگوریتم جداسازی زیردرخت)

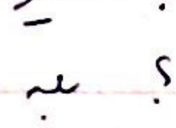
- یه block size تعریف کن. (مثلاً در اسلاید : 4)
- درخت را به روش اول ریشه بعد چپ بعد راست (pre-order) پیمایش کن.
- در صورتی اون بخش درخت را جدا می‌کنیم که: $\frac{b}{2} \leq$ تعداد entry هاش $\leq b$ قرار بگیره. ($[\frac{b}{2}, b]$)

← از ریشه شروع کنیم، value = 12 : $2 \leq 12 \leq 4$ ؟ خیر ← برو شاخه چپ ↺

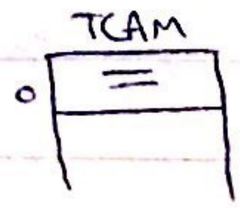
$2 \leq 6 \leq 4$ ؟ خیر ← شاخه چپ نداره پس برو راست ↺

$2 \leq 5 \leq 4$ ؟ خیر ← برو شاخه چپ ↺

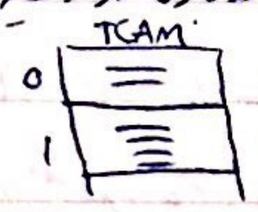
$2 \leq 2 \leq 4$ ؟ بله ⇐ پس



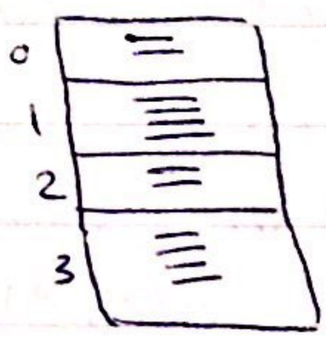
را جدا می‌کنیم می‌ذاریم در یک بلاک یعنی این 2 تا entry که این جا بوده، می‌ذاریم در حافظه‌ی TCAM. (4 تا بوده، 2 تاش هم خالی می‌مونه.)



← اینو که حذف کنیم از درخت، value ها update می‌شه، دوباره از ریشه شروع می‌کنیم، 4 تای شاخه‌ی سمت چپ ریشه در TCAM، جدا می‌شه. ←



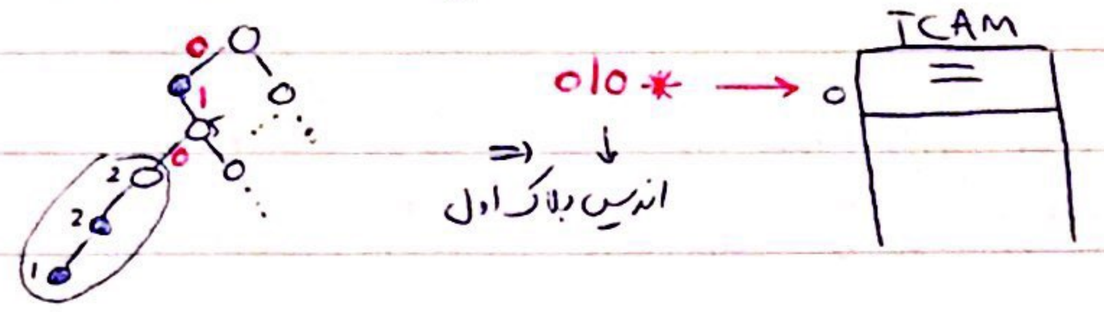
← ...

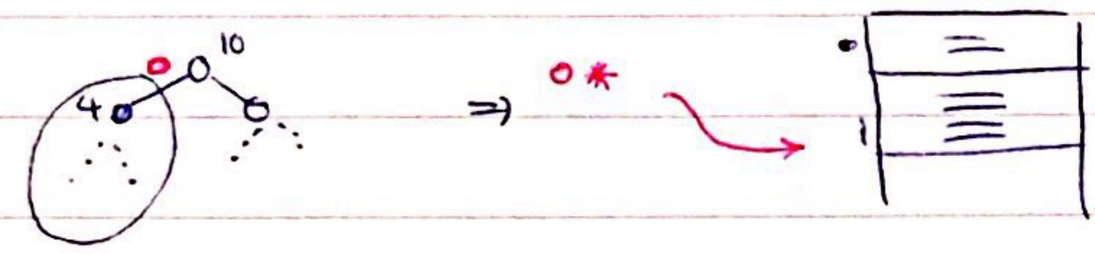


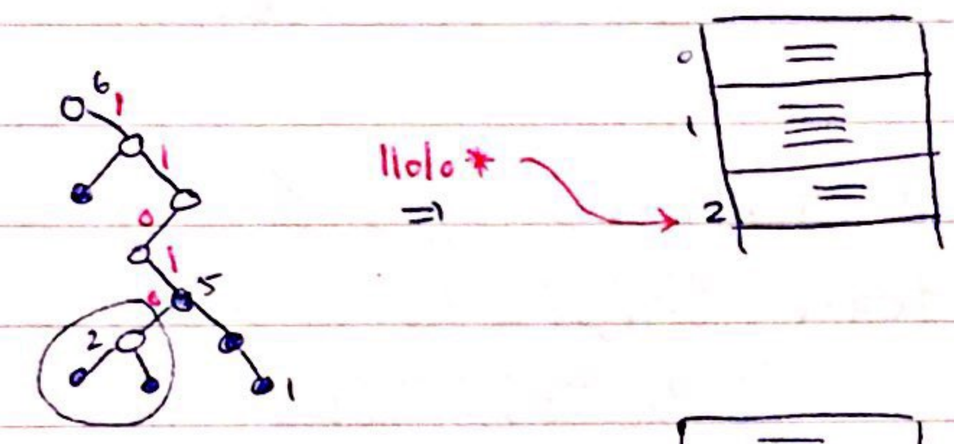
← در نهایت اندازه‌ی هر بلاک بین $\frac{b}{2}$ تا b می‌شه. تقریباً 75 درصد حافظه مفید مصرف می‌شه.

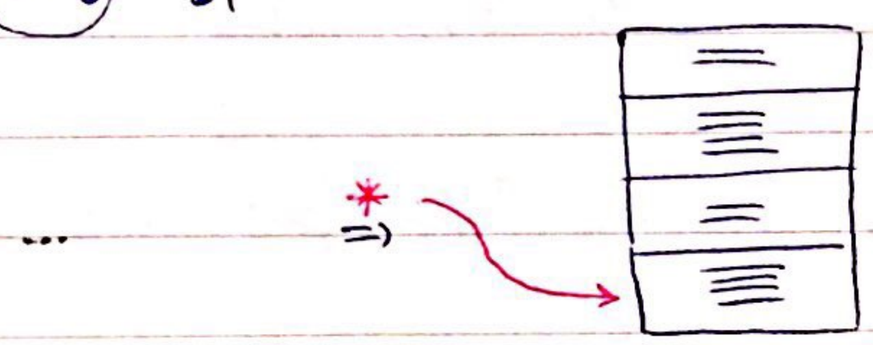
$\frac{b}{2}$ ↓ ٪50 ، b ↓ ٪100 ← متوسط : ٪75

- برای هر کدوم از اینا باید index بذاریم. نگاه می‌کنیم شاخه‌ای که جدا کردیم، در پیمایش چی بود:









p.66 : پس حافظه 4 بلاک شد. اندازه‌ی هر بلاک 2, 4, 2, 4 است. در بخش entry های bucket prefixes که در بلاک‌ها ذخیره کردیم مشخص شده. به ازای هر بلاک هم یه index داریم در ستون index.

- حالا یه آدرس شبکه‌ی مقصد که بیاد، مثلاً 01001011011 ، می‌خواهیم بگردیم، با index مقایسه می‌کنیم، با 010* match می‌شه، آدرس را در اون بلاک جستجو می‌کنیم.

- در این روش، از TCAM ، 100 درصد استفاده‌ی مفید نمی‌کنیم چون بلاک‌ها بین $[b, \frac{b}{2}]$ پر می‌شوند. حدوداً 75 درصد استفاده‌ی مفید. چون حافظه‌ها هم گران هستند، 25 درصد هدر رفتن زیاده.

2) روش Post-order split Algorithm :

p.67 : post-order می‌گه اول شاخه‌ی سمت چپ، بعد راست، بعد ریشه. ← روش پیمایش

- در اسلاید : $b=4$

- در این روش، تعداد prefix هایی که از درخت جدا می‌کنیم باید دقیقاً 4 باشه. (نه بین $\frac{b}{2}$, b)

<= $b=4$ ؟ خیر، بزرگتر از 4 است ← چپ که نداره پس برو راست

$5=4$ ؟ خیر ← برو چپ

$2=4$ ؟ خیر ← تا این جا 2 تا ← برو شاخه‌ی راستی

$2+3=4$ ؟ خیر ← برو چپ

$2+3=4$ ؟ خیر ← برو چپ

maxnote
Scanned with CamScanner

$2+2=4$ ؟ خیر ← تا این جا : 3 ← برو راست

$3+1=4$

⇐ این 4 تا (شکل a) را جدا می‌کنیم می‌گذاریم در بلاک حافظه.

← اینو حذف می‌کنیم، update می‌شه، ...

← این جا می‌شه 3 تا بلاک. هیچ هدر رفتی در TCAM نداریم. حالا باید index درست کنیم.

- چون شاخه‌ها متفاوتند، برای مثلاً بلاک اول، مجبوریم تا index درست کنیم:

010* , 01100* , 0101*

p.68 : پس 3 تا بلاک داریم. این جا index TCAM، 7 تا index داره. (قبلی 4 تا بود)

مزیت نسبت به قبلی : حافظه‌ی کمتری در TCAM مصرف می‌کند

اشکال : index TCAM بزرگتر می‌شه.

← بین این 2 باید ببینیم کدوم روش بهتر است.

p. 69 : در روش trie-based table partitioning :

پیچیدگی جستجو : $O(1)$ (2 بار مراجعه به حافظه برای پیدا کردن index ← مراجعه به index TCAM, RAM

2 بار هم مراجعه به TCAM که یکی برای مراجعه به TCAM برای entry و یکی به RAM)

حجم حافظه‌ی مصرفی : $O(N)$

زمان به روزرسانی و درست کردن index TCAM : در هر بار باید درخت پیمایش شه :

روش subtree : $O(\frac{NW}{b}+N)$

روش postorder : $O(\frac{NW}{b}+N)$

- مرور : روش های سخت افزاری جستجو در جدول forwarding را دیدیم. روش های DIR و روش های مبتنی بر TCAM.

روش های مبتنی بر TCAM به نسبت روش های مناسب تری بودند تنها مشکلی که داشتند، مصرف انرژی بالا بود که با تکنیک indexing سعی کردیم محتوای TCAM را بخش بندی کنیم و هنگام جستجو فقط یک بخش از حافظه‌ی TCAM را فعال می‌کردیم که انرژی مصرفی کاهش یابد. ← اون بخش که برای index، از روی IP شبکه، index را پیدا می‌کردیم و همون بخش را فعال می‌کردیم.

p.70 : یک تکنیک دیگه بر مبنای TCAM را می‌خواهیم توضیح دهیم و سرعت را باز افزایش دهیم. یعنی بیایم TCAM ها

موازی استفاده کنیم و با انجام عملیات جستجوی موازی، سرعت lookup IP را افزایش دهیم:

TCAM-based distributed parallel lookups:

این روش با استفاده از تکنیک bit selection میاد جدول forwarding را به 16 قسمت تقسیم می‌کند. این 16 قسمت را در TCAM های موازی قرار می‌دهد به گونه‌ای که سعی می‌کند سرعت جستجو را چندین برابر کند.

در روش قبلی با یکبار مراجعه به حافظه با O(1) می‌توانستیم lookup را انجام دهیم. اگر رسیدیم به این جا که یکبار مراجعه به حافظه هم می‌تونه زمان زیادی برای ما باشه، می‌توانیم با این تکنیک سرعت را افزایش دهیم یعنی با هر بار مراجعه به حافظه، k تا lookup انجام دهیم.

- چطوری این کار را می‌توانیم انجام دهیم؟

در این مقاله اومده 4 تا روتر را در نظر گرفته، براساس بیت های آدرس شبکه‌های مقصدی که در جدول forwarding ذخیره شده، بیت های 10، 11، 12، 13 را انتخاب کرده و نشون داده براساس این 4 بیت، جدول forwarding تقریباً به 16 گروه هم اندازه تقسیم می‌شه.

در شکل، ستون ها اندازه‌ی هر گروه هستند. گروهی که بیت های 10، 11، 12، 13 شان 0 باشه، می‌شه گروه 0 تا ... گروه 15. جدول forwarding در 4 تا روتر، به 16 گروه تقریباً مساوی تقسیم شده.

=> میایم با استفاده از روش bit selection، index را می‌گذاریم بیت های 10 تا 13 آدرس شبکه‌ی مقصد و براساس این 4 بیت، جدول forwarding را به 16 گروه تقسیم می‌کنیم.

اگر بسته‌ی دریافتی، 4 تا بیتش 0 بود، در گروه 0 را جستجو می‌کنیم و ...

p.71: نکته اش اینه می‌گه من همه‌ی اینارو در حافظه‌ی TCAM نمی‌گذارم. در k تا حافظه‌ی TCAM قرار می‌دهم. هر حافظه را به n تا partition تقسیم می‌کنم. => مجموعاً nxk تا partition داریم.

جدول میریایی را هم که به 16 تا partition تقسیم کرده بودیم. به گونه‌ای این کار را انجام می‌دهیم که:

$n \times k > 16$

باشه. بعد نگاه می‌کنم براساس جستجویی که در جدول می‌شود، اون partition هایی که بیشتر بهش مراجعه می‌شود را در حافظه های TCAM جداگانه تکرار می‌کنیم. (مثلاً پارتیشن 0 را در TCAM 1 و 2 تکرار کرده) چند تا تکرار می‌کنم؟ بستگی داره چه قدر بهش مراجعه صورت می‌گیرد.

چرا این کار را انجام می‌دهم؟ چون وقتی داره بسته‌ها پشت سر هم وارد روتر می‌شود، اولین بسته را که برمی‌دارم براساس این 4 بیت مشخص می‌کنم کدوم partition را باید جستجو کنم؟ مثلاً پارتیشن 0، حافظه‌ی TCAM 1 را (که ID=0 داره) فعال می‌کنم جستجو را انجام می‌دهم.

این که داره انجام می‌شه، بسته‌ی دوم که میاد، مثلاً ID=11 است، حافظه‌ی دوم را فعال می‌کنم. یعنی به طور موازی

می‌توانم جستجو در 2 حافظه‌ی TCAM را انجام دهم.
پس کاری می‌کنم این k تا حافظه‌ی TCAM به طور همزمان، k جستجو را انجام بدهد.
=> سرعت جستجو k برابر می‌شه.

p.72: مدار سخت افزاری که برای این روش طراحی شده را می‌بینیم.
سمت چپ: آدرس مقصد که از روی بسته برداشتیم، چهار بیت 10 تا 13 را استفاده می‌کنیم که مشخص کنیم کدوم حافظه را فعال کنیم و کدوم partition داخل حافظه را فعال کنیم.
براساس مدار کنترلی که indexing را انجام می‌دهد، مشخص می‌کنیم که بسته‌ی وارد شده، برای اینکه کام بعدش تعیین بشه، آدرسش در کدوم یکی از صف های 1 تا k قرار می‌گیرد.
مثلاً آدرس میاد در صف 1 قرار می‌گیرد، بعدی در صف 2 الی آخر. با این مدار کنترلی (indexing logic) که داریم، سعی می‌کنیم که طول صف هامون با هم برابر باشه: به گونه‌ای که اگر بسته‌ای وجود دارد، همیشه k تا حافظه‌ی ما فعال بشه. (یک partition از هر کدوم فعال باشه) ← یعنی k جستجوی موازی انجام دهیم.
نتیجه‌ی next hop مشخص می‌شه و اعمال می‌کنیم.

(محتوای حافظه ها یکسان نیست. مثلاً اگر در شکل ص 71، n=4 باشه، k=8، 32 تا partition داریم که 16 تا IP را می‌خواهیم توش بذاریم. => بعضی IP ها را می‌شه 2 یا 3 بار تکرار کرد مثلاً)

p.73: این روش را توضیح داده که با استفاده از حافظه‌ی TCAM ـ 133 MHz، توانسته، 533 میلیون بسته در ثانیه را lookup کنه که اگه طول متوسط بسته ها را در نظر بگیریم می‌شه یه لینک 160Gbps. امروزه لینکی با این سرعت هنوز نداریم.

ـ در IPv6 می‌دانیم اندازه‌ی فیلد آدرس بسته ها، از 32 بیت تبدیل می‌شه به 128 بیت. یعنی اگه درخت بایتری برویم، عمق درخت می‌شه 128. اگر بخواهیم حافظه‌های TCAM استفاده کنیم، هر کلمه‌ی حافظه می‌شه 132 بیت و یک مقایسه کننده‌ی 132 بیتی نیاز داریم که خودش پیچیدگی را به مراتب زیاد می‌کند.
=> این روش هایی که توضیح دادیم، ممکنه برای IPv4 کارآمد باشند، اما برای v6، احتمال زیاد کارایی بسیار کمی پیدا می‌کند => باید تدابیری برای IPv6 داشته باشیم.

p.75: هدر IPv6 را می‌بینیم که تعداد فیلد هاش از IPv4 کمتره. ← هدر ساده‌تر شده. چون سعی کردن در IPv6 پردازش گره های میانی کمتر بشه. ولی اندازه‌اش بزرگتره. (فیلد های آدرس مبدأ و مقصد 128 بیتی شده)

- برای حالت گذرا از IPv4 به IPv6، یک دستورالعمل برای تبدیل آدرس ها تدوین شده که می‌توانیم برقبای

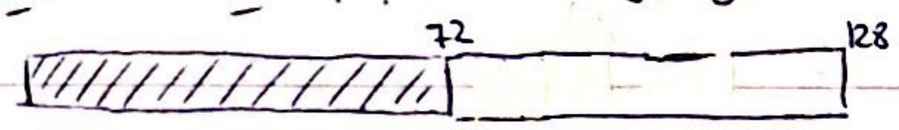
Scanned with CamScanner

اون ، آدرس های V4 را تبدیل کنیم به V6 .

P.76 : با این تبدیل که انجام می‌شه ، چیزی که در جدول forwarding مشاهده می‌شه اینه که بین 64 تا 128 ما هیچ prefix مشاهده نمی‌کنیم . اکثر prefix ها بین 48 تا 64 بیت هستند . در 128/1 تعدادی هست

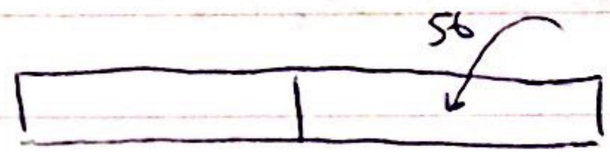
P.77 : اگر نموداری بخواهیم نگاه کنیم ، وقتی می‌آییم یک جدول IPV4 را با اون دستور تبدیل می‌کنیم به IPV6 ، چنین توزیعی پیدا می‌کند براساس تعداد بیت .
بین 32 تا 48 اکثر prefix ها قرار می‌گیره . بین 64 تا 128/1 هیچ prefix وجود ندارد .
در 128/1 تعدادی هست . بین 48/1 تا 64/1 تعداد خیلی کمی هست . تعداد خیلی کمی هم زیر 32 .
=> روش را باید متمرکز کنیم روی بازه‌ی 32/1 تا 48/1 . وادی نواحی 64/1 تا 128/1 که خالی است هم فکر کنیم .
مثلاً اگر درخت بخواهیم درست کنیم ، یه درخت درست می‌کنیم که تا 64 prefix داره ، از 64 تا 128 یه شاخه‌هایی پیمایش می‌شه هیچ گره‌ای وجود ندارد . ← شاخه طولانی ← باید در طراحی اینارو لحاظ کنیم .

P.78 : روشی که از TCAM استفاده می‌کنه : در این روش به جای اینکه از TCAM ـ 128 بیتی استفاده کند ، میاد 2 تا TCAM ـ 72 بیتی استفاده می‌کنه ؛ یعنی 2 سطحی عمل می‌کند . (به جای یک 128 بیتی)
میاد اون آدرس 128 بیتی را 72 بیت اول را جدا می‌کند :



X چون اکثر آدرس‌ها زیر 48/1 بودن پس بخش عمده از آدرس‌های مقصد زیر 56 بیت دارند .
اگر مقصد را این سطح پیدا کردیم next hop را استفاده می‌کنیم ، اگر پیدا نکردیم میاییم کل 128 بیت را جستجو می‌کنیم .
برای آدرس‌های بالای 56 بیت اول رشته‌ی مشترک را در TCAM یک را ول می‌گذاریم .
یه رشته 56 بیتی مقصد بود ، چون 56 بیت بیشتر را خالی می‌کنیم اون 72 بیت ، این 56 بیت را شامل 128 بیت IPV6 را است 56 بیت tag به یه اطلاعات next hop هم بیشتر اضافه می‌کنیم می‌شه 56 بیت . X

← 72 بیت اول را برمی‌داریم می‌ذاریم در TCAM . با توجه به شکل ص 77 ، در 72/1 همه را تقریباً پوشش می‌ده فقط تعداد کمی باقی می‌ماند .
=> 72 بیت آدرس مقصد را برمی‌داریم سرچ می‌کنیم ، اگر match شد ، مراجعه می‌کنیم به حافظه‌ی SRAM ، در SRAM که متناظر با TCAM است ، قبلاً در SRAM ، next hop را نگه می‌داشتیم ، حالا در سطح اول ، next hop را داریم و 3 تا بیت و یک tag (tag ـ 16 بیت)



این 16 بیت را از 72 بیت که مونده بود کم کنیم ، 56 بیت می‌مونه :
اگر next hop داخلش باشه (P) Primitive ادامه پیدا کرده باشه (m) و این 72 بیت Valid (v) باشه ،
اطلاعات next hop را برمی‌داریم .
از این 3 تا بیت تشخیص می‌دهیم تا الان longest prefix matching ، این next hop هست یا نیست و آیا باید

ادامه بدهیم در حافظه‌ی دوم یا نه.

=> next hop را به دست میاریم، اون 16 بیت (tag) را برمی‌داریم با اون 56 بیت، می‌شه 72 بیت می‌ریم در حافظه‌ی سطح دوم سرچ می‌کنیم یا next hop را به دست میاریم که اون LPM است یا next hop را در مرحله‌ی اول به دست میاریم.

← خلاصه: یه آدرس 128 بیتی از روی هدر بسته برمی‌داریم، 72 بیت اول را برمی‌داریم در TCAM اول جستجو می‌کنیم، از این ممکنه next hop به دست بیاد، اگه نیاد، اون 16 بیت را برمی‌داریم با 56 بیت باقی مانده در TCAM دوم جستجو می‌کنیم، از SRAM، next hop ها به دست میاریم.

(از کجا بفهمیم next hop را استفاده کنیم یا نه؟ از ترکیب اون 3 بیت ← باید در TCAM دوم هم جستجو کنیم یا نه)

← به جای TCAM 128 بیتی از 72 بیتی استفاده کردیم => کلی مصرف انرژی را کاهش داریم. چون مقایسه کننده‌ها از 128 بیتی شده 72 بیتی. اندازه‌ی 2 تا حافظه هم بار بر جدول forwarding می‌شه.

P.79 : تحلیل: در forwarding table ما N تا entry داریم. میایم این N تا رو prefix های 72 بیتی‌شون را پیدا می‌کنیم و اونایی که 72 بیت مشترک دارن ← S درصد (1 > S)

(1-S) هم، بزرگتر از 72 بیتی‌ها ست. => پس 2 تا مجموعه داریم: G_S : مجموعه‌ای که prefix های کوچکتر از 72 بیت داره. G_L : مجموعه‌ای که prefix های بزرگتر از 72 بیت دارد.

حافظه‌ی مصرفی مجموعه‌ی اول: $72 \times SN$

به علاوه‌ی درصدی از مجموعه‌ی دوم که 72 بیتش را باید در حافظه‌ی اول نگه داریم: $72(1-S)N$

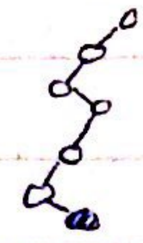
به علاوه‌ی مجموعه‌ی دوم: $(1-S)\frac{N}{\beta}$

جمعش ← $72N(1+(1-S)/\beta)$ bits. چون S نزدیک به 1 است، 1-S نزدیک صفره. پس حافظه‌ی مصرفی تقریباً $72N$ است. درصورتی که اگر از یک سطح TCAM استفاده می‌کردیم می‌شد $128N$.

=> اندازه حافظه‌ی مصرفی و مصرف انرژی را کاهش دادیم. (به نسبت $\frac{128}{72}$)

- از نظر کارایی، مرتبه‌ی جستجو هنوز $O(1)$ است. درسته 2 بار به TCAM مراجعه می‌کنیم ولی عدد ثابته. زمان جستجو کمی بیشتر می‌شه ولی مرتبه‌اش همون $O(1)$ است.

P.80 : در روش استفاده از Trie و binary trie، وقتی عمق درخت می‌شه 128، فضای جستجو 2^{128} است که عدد بسیار بزرگی است. اما وقتی اینو در قالب یک درخت باینری می‌کشیم می‌بینیم شاخه‌های خالی زیاد داریم. مثل: مثلاً این مسیر را با روش path compression فشرده کنیم و حافظه‌ای برایش مصرف



Scanned with CamScanner

نکنیم. => روش پیشنهادی: Variable-Stride Path & Bitmap Compression

- اومده اون درخت بزرگ را به یه سری subtree تقسیم کرده. این زیردرخت ها را بعد اومده در قالب bit compression نشون داده => هر زیردرخت می‌شه یک node ـ درخت جستجو.

- چطوری این کار را انجام می‌دهیم؟

p.81: هر زیردرخت، یک نود وارده داره، تعدادی نود خارجه. کاری که می‌کنیم اینه که در این زیردرخت که اطلاعات nexthop قرار می‌گیرد (مثل C) بیایم اطلاعات next hop یا pointer هایی که به جاهای دیگه هست در یک pointer array نگه می‌داریم، اونایی که تکرار شده (مثل NIP2) حذف می‌کنیم (شبیه DIR 16-16) نهایتاً اون bitmap، compressed ptr را نگهداری می‌کنیم.

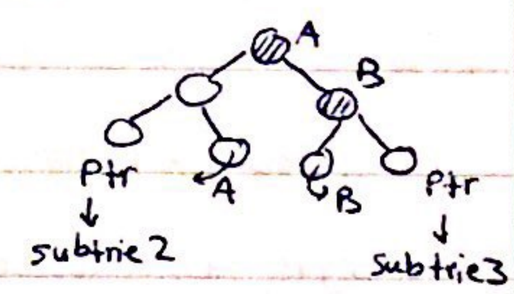
p.82: این ساختار گره‌ی ما هست. وقتی بهش می‌رسیم، عمق زیردرخت را مشخص می‌کنه، چندبیت را Skip کردیم (فشرده کردیم)، ترکیب بیت‌های Skip شده چی بودن (0 و 1 چی بودن) ← Segment، بعد bit map را قرار می‌دهیم و بعد اون next hop ها یا ptr ها.

=> در شکل ص 80، هر زیردرخت را با یک چنین ساختاری نشون می‌دیم که وقتی رسیدیم به یه زیردرخت ✗. چند بیت را Skip کردیم در مسیر؟ مثلاً 3 بیت. این 3 بیت چی بوده؟ مثلاً 010. عمق زیردرخت مورد نظر چه قدره؟ مثلاً 5 بیت. این 5 بیت چند تا ptr باید داشته باشه؟ 2^5. پس یک bitmap ـ 32 بیتی درست می‌کنیم که به تعداد 1 هاش ptr یا next hop خواهیم داشت.

=> در Segment، به تعداد $2^{\text{عمق}}$ بیت داریم. به تعداد 1 های درون bitmap هم یا ptr داریم به زیردرخت بعدی یا اطلاعات nexthop داریم.

p.83: مثال: جدول forwarding تبدیل شده به درخت binary. 3 تا نود تعریف شده که ساختارشون را در ص 84 می‌بینیم:

A: عمق 2، هیچ بیتی Skip نشده. چون عمق 2 بوده، bitmap، 4 بیت دارد. هر 4 تاش مقدار دارد.

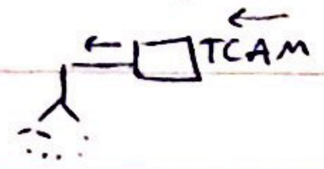


بعد Ptr subtrie2 به subtre2 اشاره می‌کنه و ...

برای رفتن به subtrie2، چندبیت را Skip کردیم؟ 3 بیت.

maxnote
Scanned with CamScanner

p.85 در نهایت می‌شه روش‌های ترکیبی هم پیشنهاد داد. بگیم مثلاً یک threshold درنظر بگیریم مثلاً k بیت. اگر k bit است از TCAM یا binary CAM استفاده کنیم، بعد ادامه‌اش را بریم در یک درخت جستجو، جستجو کنیم. پس می‌شه 2 مرحله‌ای انجام داد مرحله‌ی اول در TCAM، دوم درخت جستجو



p.87 : یک مقایسه بین روش‌های جستجو در میریاب وجود دارد که TCAM بهترین کارایی را دارد از نظر جستجو و حافظه‌ی مصرفی. فقط بحث مصرف انرژی داشت که با روش‌های گفته شده مدیریت می‌شه.

packet classification اسلاید

- دسته‌بندی بسته‌ها یکی از مباحث مهم در میریاب‌های IP است.

pkt classification به منظور تشخیص جریان‌های ترافیکی انجام می‌شود.

جریان ترافیکی: بسته‌هایی که متعلق به یک app هستند که به طور متوالی در شبکه ارسال می‌شوند تا به مقصد منتقل شوند، یک جریان ترافیکی هستند.

- تشخیص جریان ترافیک می‌تونه در سطوح مختلف انجام بشه. مثلاً جریان ترافیک بسته‌های صوتی، بسته‌های multimedia یا بسته‌هایی که از یک مبدأ به یک مقصد هستند، یا بسته‌های ترافیک web و ... .

(در سطوح مختلف یعنی مثلاً از یه app به یه app است یا یک دسته‌ای از ترافیک است که یک مشخصه‌ی یکسانی دارد. مثلاً همشون را پروتکل http تولید کرده.)

- چرا تشخیص جریان‌های ترافیکی مهم است؟ قبلش توضیح یک classifier را ببینیم:

p.2 : برای تشخیص جریان‌های ترافیکی، اول باید جریان‌ها را توصیف کنیم. توصیف جریان‌های ترافیکی با classifier انجام می‌شه.

پس ما یه جدولی به نام classifier داریم که توی اون جدول جریان‌های ترافیکی را توضیح می‌دهیم. هر سطر این جدول که بهش یک rule یا قاعده گفته می‌شه، یک جریان را توصیف می‌کند.

هر جریان ترافیکی براساس اطلاعاتی که در هدرهای لایه شبکه، لایه transport و لایه app ممکنه باشه توصیف می‌شه.

↳ protocol, dst ↳ src, dst

به عنوان مثال، اگر مقصد یک بسته 128.238/16 باشه و پروتکل لایه 4، TCP باشه و app هم telnet باشه، این یک جریان ترافیکی است که با R_4 تعریف شده. معمولاً در classifier می‌گیم چه action ی روی اون بسته انجام بده. گفته این بسته را deny کن.

الان تشخیص این جریان را به چه منظور انجام دادیم؟ به منظور pkt filtering یا کنترل دسترسی. این یک rule ی امنیتی است. می‌گیم نمی‌خواهیم در این اوتر به این شبکه مقصد کسی از بیرون telnet کنه.

Scanned with CamScanner

R2 می‌گه اگه بسته‌ای به Net D بره، مبدأش هم Net B باشه، اگه پروتکل app ش real time protocol بود، از پورت 3 خارج کن تا از طریق یک شبکه سرعت بالا بره به مقصدش.
یه جور مسیریابی براساس سیاست گذاری (policy routing) انجام می‌دهیم.
چرا؟ چون احتمالاً این شبکه ATM سرعت بالا، delay کمتری دارد.

R3 می‌گه اگه مقصد Net B باشه، مبدأ Net C، اگه نرخ ترافیک از 10 Mbps بیشتر شد، drop کن.
داره traffic policing انجام می‌ده.

- پس چرا باید جریان های ترافیکی را تشخیص دهیم؟
1. برای اینکه می‌خواهیم pkt filtering کنیم برای مباحث کنترل دسترسی و مباحث امنیتی.
2. می‌خواهیم مسیریابی بر مبنای سیاست گذاری انجام دهیم برای ارائه‌ی بهتر QoS. ← اعمال سیاست های مدیریتی در شبکه
3. می‌خواهیم مدیریت ترافیک کنیم در شبکه از ازدحام جلوگیری کنیم traffic policing انجام دهیم به منظور ارائه بهتر خدمات و سرویس دهی.

- جریان ها را چگونه تشخیص می‌دهیم؟ از اطلاعات هدر لایه شبکه و انتقال
جریان ها را چگونه توصیف می‌کنیم؟ در قالب rule های درون classifier. هر rule شامل یک توصیف از جریان هست براساس فیلدهای هدر و همچنین شامل یک action که باید روی بسته اعمال شود.

R4 می‌گه از هر مبدأ به هر مقصد هر پروتکلی هست بپذیر. هر بسته بیاد با R4، match می‌شه ولی اگه با R1 یا R2 یا R3 هم match بشه چی؟ پس یک اولویت گذاری بین rule ها لازمه که اگر یک بسته با چیزی تا از rule های classifier مطابقت کرد، کدوم اعمال بشه؟
معمولاً در classifier، ترتیب اولویت است. ($R_4 > R_3 < R_2 < R_1$)

P.3 : نحوه‌ی عمل: یک classifier داریم که rule ها را توش توصیف کردیم، بسته‌ای که وارد می‌شه، هدرها را سر می‌داریم، با استفاده از Classifier، pkt classification انجام می‌دهیم. نتیجه اینکه یک action را پیدا می‌کنیم که روی بسته اعمال کنیم.

P.4 : هر rule در classifier داره یه جریان ترافیکی را توضیح می‌ده که این جریان شامل یک regular exp. هست، از فیلدهای هدر مثل آدرس مبدأ و مقصد و... گرفته می‌شه. فرض کنیم d فیلد هدر را استفاده کردیم برای توصیف یک جریان.
اولویت را مشخص می‌کنیم (معمولاً ترتیب). و action که باید روی بسته اعمال بشه.

maknote

Scanned with CamScanner

یک بسته‌ی وارد شده، اون d فیلد هدرش را بررسی داریم. نگاه می‌کنیم با کدوم rule ها مطابقت داره، اون که بالاترین اولویت را دارد، انتخاب می‌کنیم و بعد action را روی بسته اعمال می‌کنیم.

P.5 : مثالی از یک classifier که از 4 فیلد هدر استفاده کردیم. بعضی از فیلد ها به صورت محدوده تعریف می‌شوند (*00 یا *110). بعضی از فیلدها یک عدد هستند، مثل port number. ((4,8) می‌شه 4 یا 8). برای هر rule یک action داریم.

P.6 : یک روش pkt classification چه ویژگی هایی باید داشته باشه؟ معیارهای کارایش چیاست؟ (چه روشی خوبه؟)

1) سرعت جستجو : هر چی سریعتر بتوانیم classification انجام دهیم روش بهتری است.
2) هزینه : با حافظه‌ی مصرفی بیان می‌شه که چه قدر مصرف می‌کنه.
- از نظر مقیاس پذیری 2 تا شاخص داریم. classifier یک جدول دو بعدی است. یعنی هم می‌تونه تعداد فیلد ها زیاد شه هم تعداد rule ها. ممکنه یک روش در یک بعد scalable باشه در بعد دیگه نباشه. پس 2 تا معیار داریم
3) مقیاس پذیری در اندازه‌ی classifier ← (با بزرگ شدن، سرعت جستجو چه تغییری می‌کنه؟ حافظه مصرفی؟)
4) مقیاس پذیری در تعداد فیلدهای هدر ← (اگه فیلدهای هدر را زیاد می‌کنیم، سرعت جستجو چه قدر افزایش؟ حافظه مصرفی؟)
5) زمان به روز رسانی classifier
6) انعطاف پذیری در توصیف جریان های ترافیکی : مثلاً ممکنه اون (4,8) را یک روش نتواند توصیف کند مجبور شویم R2 را در 2 تا rule بنویسیم.

P.7 : روش های موجود pkt classification را می‌شه در 4 گروه دسته بندی کرد :

- مبتنی بر درخت جستجوی trie
- روش های هندسی که بر اساس فضای هندسی الگوریتم ها ارائه شده. (مثلاً اگه 3 تا فیلد داشته باشیم، یک فضای 3 بعدی داریم که هر rule یک ناحیه در این فضای 3 بعدی را بیان می‌کنه :

هر بسته‌ی وارد شده، یک نقطه از این فضاست. پس با روش های pkt classification می‌خواهیم بفهمیم این بسته‌ی وارد شده، این نقطه، در کدوم rule ها قرار می‌گیرد.)
- روش های ابتکاری (heuristic) ← ولی این روش ها معمولاً چندان کاربرد ندارد چون بر اساس تجربه به دست اومده ممکنه اون ویژگی های مدنظر ما را به صورت general نداشته باشه.
- روش های مبتنی بر TCAM.

- روش های هندسی می‌تونه مثل یک چیز باشه مثلاً از ایده‌ی اون استفاده کنیم بعد بریم سراغ روش های TCAM یا درخت

Scanned with CamScanner