بسم الله الرحمن الرحيم

System Design من مستخدم إلى مليون مستخدم

شرح مختصر للإجابة عن أشهر أسئلة المقابلات، واكتساب العديد من الأفكار، لأشهر المشاكل التي تواجه الأنظمة المختلفة

تأليف: أنيس حكمت أبوحميد

الموقع الإلكتروني: 2nees.com

المقدمة

الحمد لله ربّ العالمين، يُحب من دعاه خفياً، ويُجيب من ناداه نجيّاً، ويزيدُ من كان منه حيِيّاً، ويكرم من كان له وفيّاً، ويهدي من كان صادق الوعد رضيّاً، الحمد لله ربّ العالمين.

يُعَدّ تصميم النظام أحد المراحل الحاسمة في عملية تطوير البرمجيات وبناء الأنظمة التقنية الحديثة. ويمثل هذا المجال من العلوم الحاسوبية نقطة التقاء بين الإبداع والتكنولوجيا، بحيث يتمكن المصممين من تحويل الأفكار الجديدة والمستقبلية إلى أنظمة واقعية وقوية، وقابلة للتوسع ومجاراة التحديات القادمة والبحث عن الحلول للمشاكل قبل وقوعها...

يهدف هذا الكتاب إلى إلقاء الضوء على مفاهيم متنوعة وحلول تقنية لأشهر الأنظمة والمشاكل المتعلقة بها، في محاولة عملية لاكتساب المهارات اللازمة لتلبية احتياجات المستخدمين بما تقتضيه متطلبات هذا العصر، ويتناول هذا الكتاب موضوعات متنوعة تشمل:

- آلية فهم متطلبات المشروع: سيتعلم القارئ كيفية التعامل مع المتطلبات وتحديد المشاكل المحتملة والتي يمكن أن يواجهها في مرحلة التصميم.
- تصميم الهندسة العكسية: سنلقي نظرة عميقة على كيفية تحليل الأنظمة الحالية وتحسينها وتحديثها لتلبية المتطلبات الجديدة، وذلك من خلال السير تدريجيا في بناء الأنظمة ثم التحول من شكل لآخر كما ستلاحظ في تسلسل الشرح الخاص بكل موضوع في هذا الكتاب

- النمذجة والتخطيط: سنغوص في عالم التصميم بما يتناسب مع مقتضيات الموضوع، وذلك لضمان أكبر فائدة مرجوة

ملاحظة: هذا الكتاب سيقدم الكثير من المواضيع والعناوين والمفاهيم المتنوعة، لكنه لن يغوص في التفاصيل الدقيقة لكل مفهوم أو تقنية، فهذا الكتاب يمثل نقطة البداية للتعرف على أشهر المفاهيم التقنية وليس لمناقشتها في ذاتها...

وفي النهاية، نسأل الله -سبحانه وتعالى- أن يكون هذا الكتاب مصدرًا مفيدًا لكل من يسعى لتعزيز معرفته في مجال تصميم الأنظمة وتحسين قدرته التقنية على التعامل مع المشاكل قبل حدوثها...

ملاحظة 2: يعد هذا الكتاب ملخص لكتاب System Design Interview لمؤلفه Alex Xu، ومع أن الكتاب الأصلي كتب بطريقة لتتناسب مع أسئلة المقابلات، إلا أنني آثرت هنا قدر الإمكان تحويل المفهوم الأصلي من مجرد إجابات لأسئلة المقابلات إلى طريقة حوار وتسلسل لحل المشكلات، مضافا لها بعض التعليقات أو الشروحات من خبرتي الشخصية...، وفقنا الله وإياكم لما يحب ويرضى...والحمد لله رب العالمين.

فائدة

معنى (الْحَمْدُ لِلَّهِ) : الشكر خالصًا لله جل ثناؤه دون سائر ما يُعبد من دونه، ودون كلِّ ما برَأَ من خلقه، بما أنعم على عباده من النِّعم التي لا يُحصيها العدد، ولا يحيط بعددها غيره أحدٌ، في تصحيح الآلات لطاعته، وتمكين جوارح أجسام المكلَّفين لأداء فرائضه، مع ما بسط لهم في دنياهم من الرزق، وَغذَاهم به من نعيم العيش، من غير استحقاق منهم لذلك عليه، ومع ما نبَّههم عليه ودعاهم إليه، من الأسباب المؤدِّية إلى دوام الخلود في دار المُقام في النعيم المقيم. فلربِّنا الحمدُ على ذلك كله أولا وآخرًا.

- نقلا من تفسير الطبري

الفهرس

المقدمة 2
الفهرس..... 4
الجزء الأول: من مستخدم واحد إلى مليون مستخدم 6
فاصل معرفي NoSql UseCase 31
BACK-OF-THE-ENVELOPE ESTIMATION 36
A FRAMEWORK FOR SYSTEM DESIGN INTERVIEWS 44
DESIGN A RATE LIMITER 48
Algorithms for rate limiting 51
High-level architecture 58
Rate limiter in a distributed environment 63
DESIGN CONSISTENT HASHING 66
DESIGN A KEY-VALUE STORE 70
Distributed key-value store 71
System components 75
DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS 87
DESIGN A URL SHORTENER 94
DESIGN A WEB CRAWLER 104
DESIGN A NOTIFICATION SYSTEM 124
DESIGN A NEWS FEED SYSTEM 140
ملاحظة مهمة -الفصول اللاحقة-: 155
DESIGN A CHAT SYSTEM 156
DESIGN A SEARCH AUTOCOMPLETE SYSTEM 163
DESIGN YOUTUBE 169
DESIGN GOOGLE DRIVE 175
الخاتمة 178

الجزء الأول: من مستخدم واحد إلى مليون مستخدم

ينطلق أي نظام أو تطبيق عند إنشائه من نقطة الصفر، هذه النقطة تمثل لحظة البداية وتجميع العناصر والمكونات المطلوبة لإنشاء النظام المطلوب، في هذه المرحلة يمكنك البدء ببناء العناصر والمكونات الأساسية والطبيعية لخدمة عدد مستخدمين قليل مع نظرة مستقبلية للتوسع، أي يمكنك رسم ما تحتاجه خطوة بخطوة من المكونات الأساسية وصولا إلى المكونات المعقدة والمترابطة...، هذا الأسلوب مهم جدا ومفيد جدا لبناء تخيل حقيقي وواقعي عما يراد إنجازه، ويمكن الاستفادة منه في بناء النظام بشكل سريع وسهل يخدم المستخدمين في مرحلتهم الأولى مع قدرة النظام على التوسع والتمدد ليخدم ملايين المستخدمين بمرور الزمن، وهذا فيه حفظ للوقت والمال...

والآن لنطبق مثال عملي يوضح ما ذكر أعلاه، ولنفترض أننا سنقوم بإنشاء نظام اسمه س يمكن الوصول إليه من خلال الهاتف المحمول أو من خلال الحواسيب الشخصية، ويمكن لهذا النظام أن يتوسع من مستخدم واحد إلى مليون مستخدم...
إذا، سنبدأ من نقطة الصفر:

1. أولا سنقوم بإنشاء Server، هذا السيرفر سيحتوي ما نحتاجه من خدمات مثل ال web application, cache, database إلى آخره...

2. الخطوة الثانية اختيار نوع قاعدة البيانات المناسبة، هل ستكون Sql Database أو NoSql Database، فمثلا لو كانت كمية البيانات كبيرة وليست منظمة (unstructured data) سنختار NoSql...
3. الخطوة الثالثة سنقوم بفصل قاعدة البيانات على Server منفصل، وهذا سيقدم خدمة كبيرة لنا، وهي أن ال Web/Mobile traffic ستكون على Server، وقاعدة البيانات ستكون على Server آخر، وهذا سيتيح لنا التوسع بناء على الحاجة وبما يتناسب مع ما هو مطلوب بشكل أفضل...، يمكنك تسمية ال Server الخاص بقاعدة البيانات ب Data Tier، وال Server الخاص بال Web/Mobile traffic ب Web Tire...
4. الخطوة الرابعة التفكير بآلية التوسع المناسبة ووقتها، وهنا لدينا خيارين يمكن أن نفكر فيهما الأول يطلق عليه Vertical Scaling أو Scale Up والثاني يطلق عليه Horizontal Scaling أو Scale Out، فمثلا يعد اختيار ال Vertical Scaling خيارا ممتازا في حالة ال Low Traffic، لكنه يعد خيارا سيئا إن أصبح ال Traffic الخاص بك كبيرا...، وللتوضيح، يقصد بال Vertical Scaling هو التوسع من خلال زيادة حجم المكونات مثل زيادة حجم ال Cpu أو Ram لتلبية احتياجات ال Traffic المطلوب حاليا، بينما يقصد بال Horizontal Scaling هو إضافة Servers أخرى إلى مجموعتي حتى تحتوي كمية ال Traffic المطلوبة...، وفهم هذا الأمر مهم جدا، ففي حالة ال Low traffic عند زيادتك لل CPU مثلا، فهذا سيكون حلا ممتازا إذا واجهتك مشكلة ما...، لكن لن تستطيع فعل هذا للأبد، فال hardware مهما وصل إلى إمكانيات فإنه سيكون بعيدا عن كمية المعالجات التي تتطور بشكل

طردي معه أيضا، وطبعا ليس من الممكن إضافة مثلا 1000 cpu و 1000 ram على ال server لإتمام المهام المطلوبة، وتعد هذه واحدة من محددات استخدام ال Vertical Scaling، والمشكلة الثانية التي قد تواجهك أن ال server لو حصل به أي خلل أو أصبح down فكل خدماتك لكل الأشخاص ستكون down أيضا...، لذلك في الحالات التي تتعامل مع تمدد وتوسع كبير فإننا نتجه لاستخدام ال Horizantal Scaling، بينما نكتفي بال Vertical Scaling بال Low traffic...، ومن هنا ننتقل إلى الخطوة التالية...

5. الخطوة الخامسة استخدام ال Load balancer، وهو أسلوب تقني رائع لتوزيع الأحمال بشكل أفقي، أي توزيع ال traffic القادمة من ال mobile/web على مجموعة من ال servers، وكلما احتجنا server قمنا بإضافته والحياة سعيدة ^^، وكلمحة سريعة كيف يعمل ال load balancer يمكنك تخيل الآتي، لديك traffic قادم من مستخدم، هذا المستخدم يمكنه التعامل مع Public IP address، هذا ال IP سيمثل ال load balancer server والذي سيقوم بالتحكم وتوزيع ال traffic على ال servers المختلفة، هذه ال server التي يتحكم بها ال load balancer يتم التواصل فيما بينها من خلال Private IP (وذلك للحصول على أمان أعلى)، وبناء على هذا نكون تخلصنا من المشاكل الخاصة بال Vertical Scaling، فلو أصبح server 1 down، ف server 2 سيقوم باستقبال الطلبات أو خدمة الآخرين، وإذا زاد عدد ال traffic فيمكن إضافة server جديدة، كما يمكن حذف هذه ال server إذا قل ال traffic!، من خلال هذا فقد ضمنا توسع ال Web tire، لكن ماذا عن ال Data tier؟؟

6. حتى هذه اللحظة، ما زلنا نتحدث عن قاعدة بيانات واحدة لدينا!، لذلك، فما زال لدينا إمكانية عالية لحدوث خلل وفشل قد يؤدي إلى تعطل النظام، لذلك يأتي هنا دور ال Database replication، ويقصد بال Database replication باختصار عملية نسخ البيانات الموجودة في قاعدة البيانات ومشاركتها في أكثر من مكان، بحيث يتم ضمان تحديث البيانات في جميع الأماكن، وغالبا ما يتم هذا بوجود علاقة بين ال Master (النسخة الأصلية) وبين ال Slave (النُسخ)، وبشكل عام تكون قاعدة البيانات ال Master مستخدمة لغايات ال Write فقط، وتستخدم ال Slaves لغايات ال Read فقط، ويكون عادة عدد ال Slaves أكبر من عدد ال Master لأن أغلب المواقع تحتاج إلى Read أكثر من ال Write...، لذلك معظم العمليات مثل ال insert و update يتم إرسالها إلى ال master...، وبناء على ذلك، يقدم هذا النموذج العديد من الميزات منها تحسين الأداء، فوجود مكان محدد لل read ومكان محدد لل write سيسمح بمعالجة أكثر من query بشكل parallel، وسيتم توزيع الحمل الخاص بال read حسب الحاجة ودون أن يتأثر المستخدمون بذلك بشكل سلبي، كمان أن هذا النموذج يجعلك في أمان في حال تعطلت إحدى النسخ، فوجود مكان آخر يمكن جلب البيانات منه سيجعل النظام الخاص بك قيد العمل!، وبكل تأكيد وجود هذه المزية يعني Scale Out ^^.

ملاحظة: في حال وجود قاعدة بيانات Slave واحدة فقط، ولأي سبب أصبحت Down، فإن قاعدة البيانات ال Master تصبح Read/Write لفترة مؤقتة حتى يعود ال Slave، وفي حال حدوث خلل لل Master فإن أحد ال Slaves يصبح Master، لكن هنا يجب أن نأخذ بعين الإعتبار البيانات الجديدة التي تمت كتابتها

وكيف يمكننا جعل البيانات محدثة، فمثلا من الحلول لهذه المشكلة وجود سكربت يقوم بأداء هذه المهمة أو من خلال وجود أكثر من Master database...

7. الآن لقد قمنا ببناء نظام قابل للتمدد والتوسع بشكل لطيف، لكن، يمكننا أيضا إضافة بعض اللمسات الجميلة التي يمكن أن تسرع من الوقت أو الزمن الفعلي لل Load أو ال Response، وذلك من خلال استخدام ال Cache وال CDN

a. ال Cache: يمثل ال Cache وسيلة وطريقة مميزة لحفظ البيانات بشكل مؤقت، فمثلا إذا كان لديك Response يحتاج إلى وقت طويل ليتم إنجازه أو لديك صفحة يتم طلبها بشكل كبير، يمكنك توفير الوقت الخاص بهم من خلال حفظ نسخة من ال Response بشكل مؤقت بدل من عملية الطلب من قاعدة البيانات في كل مرة، تخيل مثلا صفحة تم طلبها 100 مرة، بوجود ال Cache سيتم طلب البيانات من قاعدة البيانات وحفظ ال Response ومن ثم ال 99 طلب التالي سيكون من خلال ال Cache، وهنا نكون وفرنا تكلفة ووقت ل 99 request، أما بدون وجود Cache فسيتم عمل 100 Request و 100 Response من وإلى قاعدة البيانات...!، وبما أن هذا الأمر سهل، يمكننا أن ننتقل إلى ال Cache Tire...

ال Cache Tire هي Layer بين ال Web Server وال Database لحفظ البيانات بشكل مؤقت، مع قابليتها للتمدد والتوسع، وهذا يقدم بدوره أداء ذو كفاءة عالية وحمل أقل على قاعدة البيانات، ويضاف إلى ذلك وجود أكثر من إستراتيجية يمكنك استخدامها في التعامل مع ال Cache، منها:

i. Read-Through: في هذا الأسلوب يطلب التطبيق البيانات من ال Cache، إذا لم يكن موجودا في ال Cache Layer يقوم بجلب البيانات من قاعدة البيانات ومن ثم إرجاعها للتطبيق، ويقوم التطبيق بعدها بحفظ هذه القيمة داخل ال Cache.

ii. Cache-Aside: في هذا الأسلوب يطلب التطبيق البيانات من ال Cache، إذا لم يكن موجودا في ال Cache Layer يقوم بجلب البيانات من قاعدة البيانات ومن ثم تخزين هذه البيانات داخل ال Cache، ومن ثم يقوم بإرجاع النتائج إلى التطبيق.

iii. Write-Through: في هذا الأسلوب يقوم التطبيق بالكتابة على Cache أولا ثم يقوم بعدها بالكتابة على قاعدة البيانات.

iv. Write-Behind: في هذا الأسلوب يقوم التطبيق بالكتابة على ال Cache، ومن ثم يقوم بالكتابة داخل Queue، بحيث يتم ضبط المدة الزمنية الخاصة بالكتابة على قاعدة البيانات، مثلا لو تم ضبطها بعد دقيقة، فإن عملية الكتابة على قاعدة البيانات ستتم بعد دقيقة من عملية الحفظ على Cache...

وغيرها من الأساليب أو الاستراتيجيات التي يمكنك الإطلاع عليها...

b. ال CDN هي اختصار ل Content delivery network، وهي تمثل شبكة من ال servers في مواقع جغرافية مختلفة تقوم على إرجاع ال Static Content مثل الصور وملفات الجافا سكربت وملفات التنسيق css...إلى آخره، لكن هناك مفهوم آخر يمكن ربطه مع ال CDN، وهو Dynamic

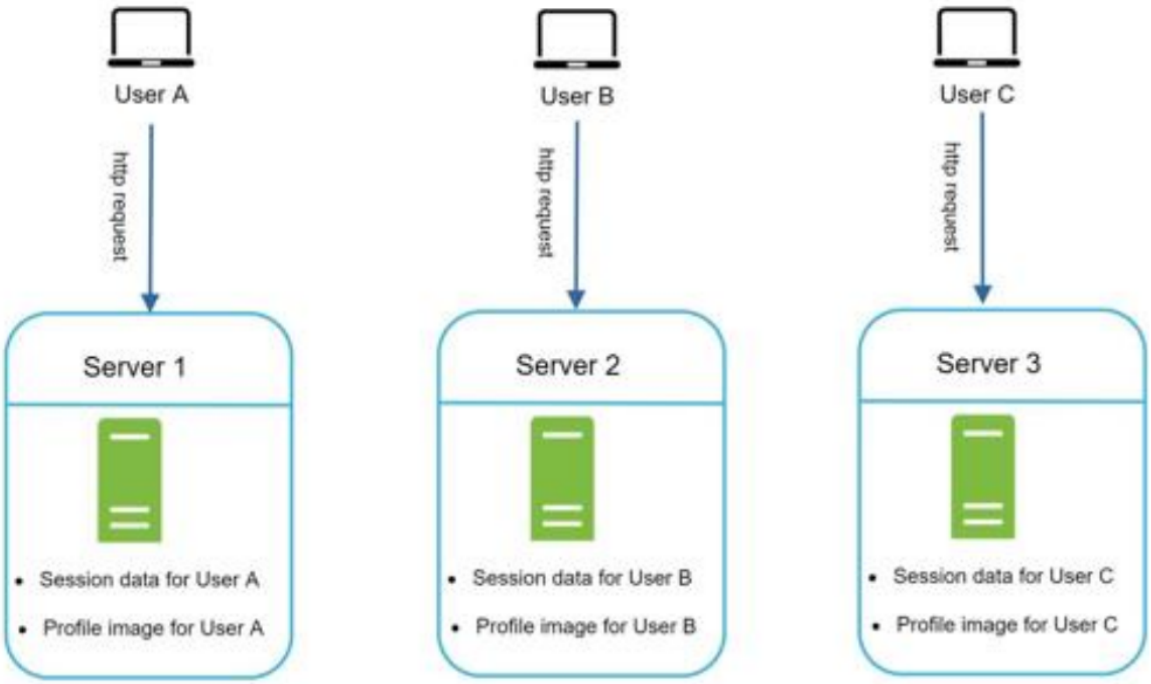
Content Caching، وهو مفهوم يسمح بعمل Cache لصفحات Dynamic يتم إنشائها من خلال ال Server ومن ثم حفظها داخل ال CDN بدلا من جلبها كل مرة من ال Origin Server، وسنتحدث هنا عن آلية ال CDN مع ال Static Content، أولا يقوم المستخدم رقم 1 بطلب صورة س من ال CDN، ال CDN لا يملك الصورة فسيقوم بطلبها من ال Origin Server، ثم سيقوم ال CDN بحفظ هذه الصورة عنده بال Cache وإرجاعها للمستخدم، إذا جاء المستخدم رقم 2 وقام بطلب نفس الصورة، فسيقوم ال CDN بإرجاعها دون الحاجة لطلبها من ال Server، ونقطة الجمال هنا في نقطتين، الأولى أنك كلما كنت قريبا من ال CDN كانت سرعة جلب البيانات أسرع، وهنا نتحدث عن سرعة جلب تقدر بأجزاء الثانية!، لكن احذر، يجب عليك اختيار الوقت المقدر لحفظ الصورة على CDN بعناية، فالوقت الطويل جدا والقصير جدا عادة ما يكون بلا فائدة مرجوة إلا في حالات محددة أو قليلة، كما أنك يجب أن تضمن الوصول إلى الأصل في حال تعطل ال CDN...

الآن وبعد كل العناوين الجميلة التي ذكرناها بالأعلى، قد تتسائل، أهذا كل شيء؟!، والجواب لا بكل تأكيد!، فهذه مجرد رؤوس أقلام، سننطلق منها لأجزاء أخرى، ولنثري ما تم طرحه آنفا، فلنذهب بشكل أكثر عمقا على ال Web Tier...

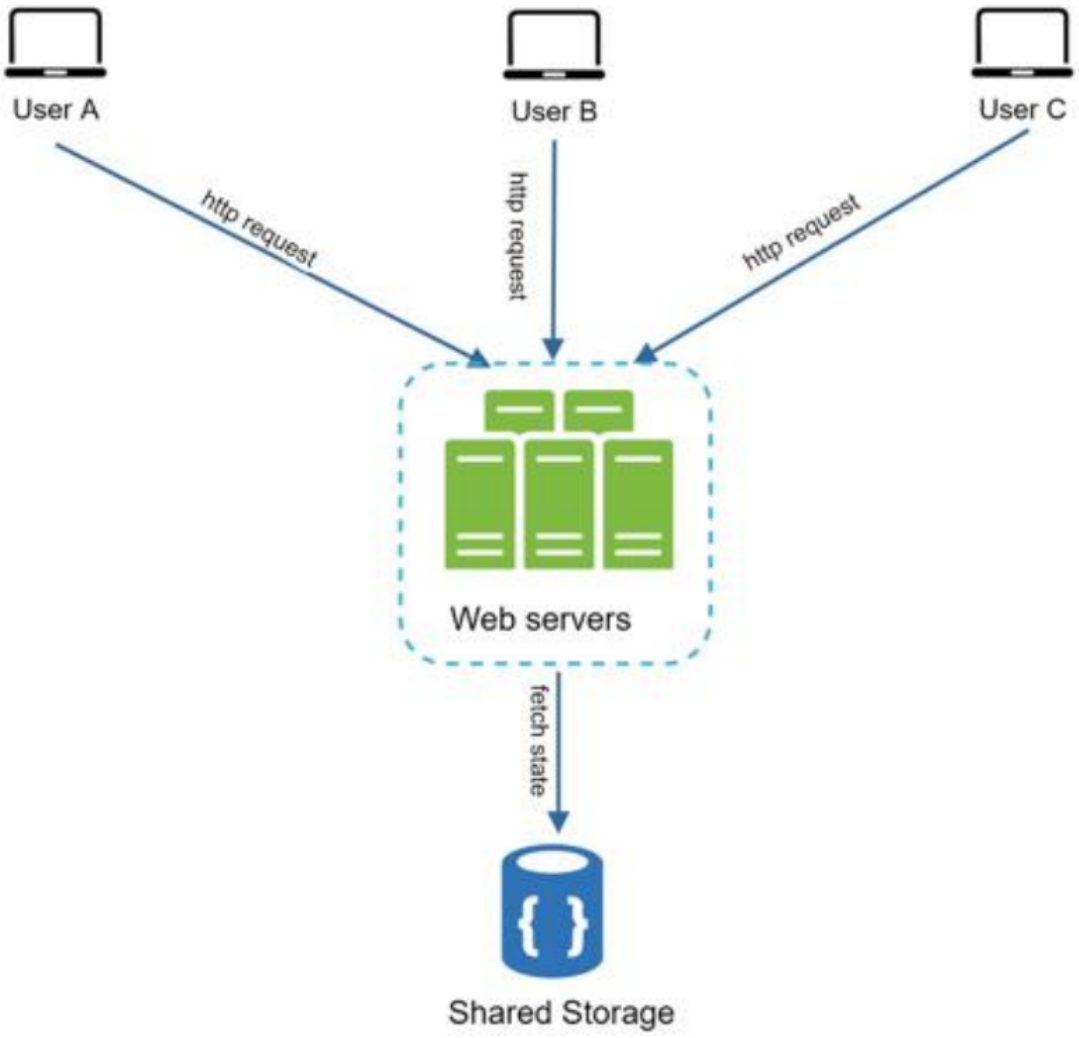
لقد تحدثنا سابقا في ال Web Tier عن إمكانية وجود أكثر من server يخدم المستخدمين، هؤلاء المستخدمين لديهم مجموعة البيانات الخاصة بهم والتي يتمايزون بها أو تستخدم للتحقق من

هويته!، ومن هذا الباب يوجد لدينا عدة مشاكل يجب أن نأخذها بعين الحسبان عند تعاملنا مع نظام يتمدد أو يتوسع بشكل أفقي، منها كيف يمكن أن نحافظ على ال Session Data وكيف يمكن أن نشارك هذه البيانات بين السيرفرات، ومن هنا يظهر لدينا مفهومان مهمان، وهما: Stateless / Stateful architecture web tier...

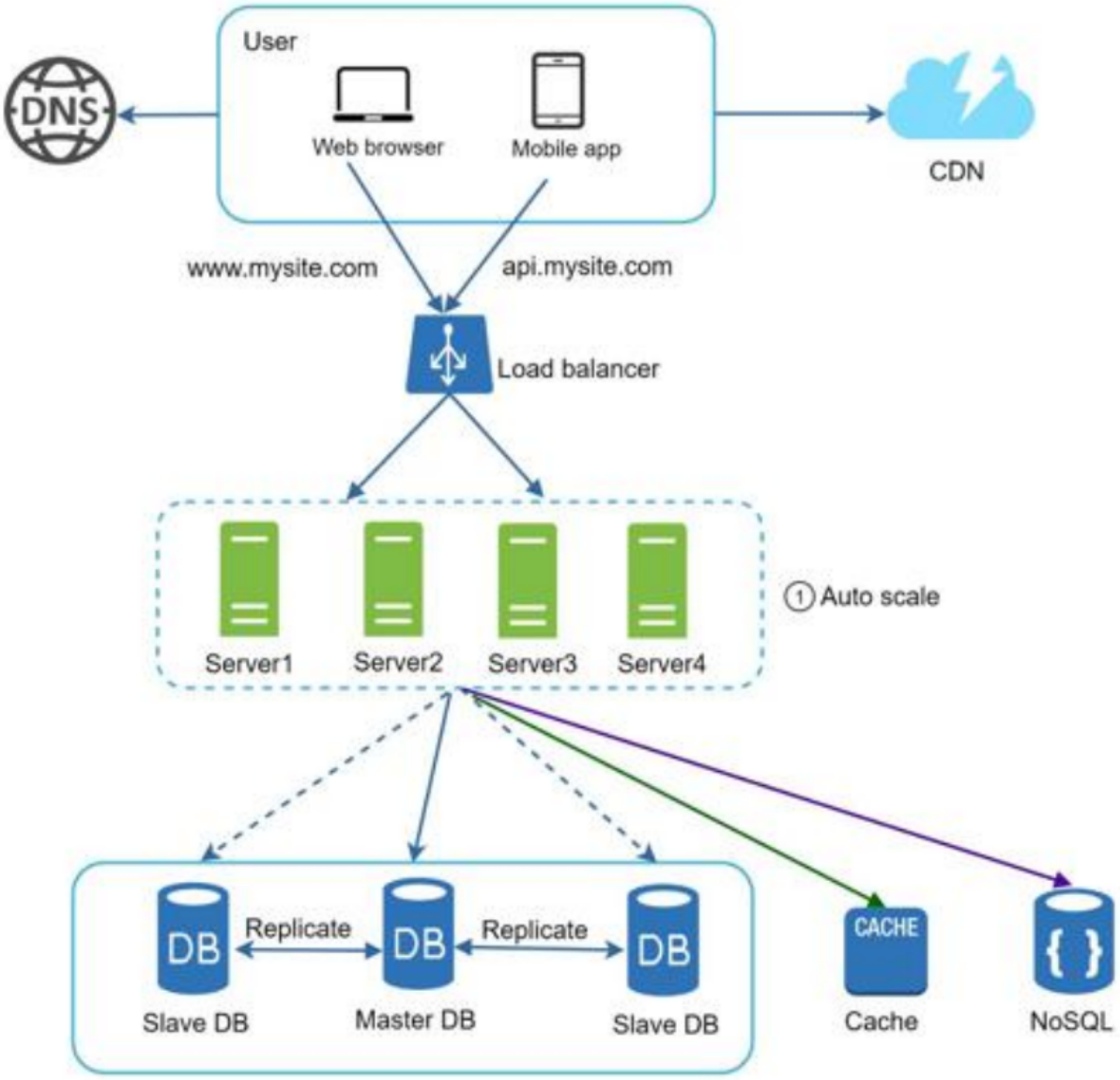
1. Stateful architecture: في هذا المفهوم يقوم السيرفر بحفظ المعلومات (Session/State) الخاصة بال Client مع كل Request خاص به، وهذا الأسلوب عادة ما نتعامل معه عند بنائنا تطبيقات على سيرفر واحد، فكل العمليات من ال login وصولا إلى أي form submit غالبا ما تتم على نفس السيرفر، لكن، في حالة وجود Web tier فهنا لدينا مشكلة، وهي أن المستخدم رقم 1 لو ذهب إلى السيرفر رقم 1، فيجب أن تبقى جميع ال requests الخاصة به واتصاله مع ال Server رقم 1، إذا ذهب Request من المستخدم رقم 1 إلى السيرفر رقم 2 فإن الخادم لن يتعرف على هذا ال Request، فهو لم يحفظ أو يجد أي Session/State خاصة بالمستخدم رقم 1 عنده...، وهنا تظهر حلول لمعالجة هذه المشكلة، لكن هذه الحلول معقدة وستزيد من صعوبة إنشاء سيرفرات جديدة، وستبني لك تحديات حقيقية عند التعامل مع أي Fail من أي سيرفر...



2. Stateless architecture: في هذا المفهوم لا يتم حفظ أي معلومات (Session) على ال Server نفسه، وإنما يتم حفظ المعلومات في مكان آخر، ومن الأساليب المميزة للقيام بهذه المهمة قواعد البيانات!، فيمكنك حفظ المعلومات الخاصة بالمستخدمين داخل NoSql أو Relational Database، بعملية الحفظ هذه يمكن لكل سيرفر من الوصول إلى المعلومات الخاصة بالمستخدم من خلال قاعدة البيانات، وفي هذه الحالة لن تهتم إلى أي سيرفر سيذهب ال Request من ال Client، فكل سيرفر سيجلب المعلومات الخاصة بهذا ال Client من خلال قاعدة البيانات هذه!، وهذا ما يسمى ب stateless web tier، ولنوضح التسلسل الخاص بهذه العملية بشكل سهل، فأولا ينطلق ال Request من عند ال Client إلى ال Servers، ثم يقوم ال Server بجلب المعلومات من ال Shared Database.


User A
User B
User C
http request
http request
http request
Web servers
fetch state
Shared Storage

وهذا هو الشكل الخاص بمعمارية ما تحدثنا عنه حتى هذه اللحظة:



لاحظ هنا أننا فصلنا ال Shared database عن ال servers، وبهذا فنحن نتحدث عن stateless web tier، استخدام ال NoSql في هذه الحالة له مميزات، فهو أسهل للتعامل عند حاجتك للقيام ب scale، عملية ال Auto-scale عملية جميلة جدا، فبها تكتمل الحلقة،

ويقصد بال auto-scale أن يزيد عدد ال server أو يقل بناء على ال traffic الخاص بالمستخدمين...، ولأننا نتحدث عن المستخدمين الذين يتواجدون في أكثر من مكان على وجه الأرض، يمكننا أن نحسن ونستغل ما قمنا ببنائه ليكون في أكثر من مكان على وجه الأرض، بحيث تكون المدة الزمنية أسرع في ال Request/Response بالنسبة للمستخدم، وهذا يمكن أن يحصل من خلال ال Data centers...

8. Data centers: تعتبر ال Data centers مكان ضخم مكون من العديد من الخوادم ومصادر الطاقة ونقاط التخزين، ولديه سرعة اتصال عالية بالإنترنت، تستخدمه المنظمات والمؤسسات لحفظ البيانات والتطبيقات الخاصة بها هناك، وتسمح للوصول إلى هذه البيانات أو المعلومات ضمن قواعد معينة، وتتمثل المكونات الأساسية لأي Data centers ب routers, switches, firewalls, storage systems, servers, application-delivery controllers، وهذا لا ينفي وجود Data centers صغيرة أو شخصية...، ولأننا تحدثنا عن مكان ضخم، فنحن نتحدث عن وجود فيزيائي، أي أننا نحتاج لمكان على أرض الواقع لهذا الغرض، لذلك، هناك العديد من ال Data centers التي تتوزع على هذه الأرض، ووجودها مفيد جدا لتحسين سرعة الاستجابة للطلبات القادمة من المستخدمين، كما أنها تخفف عبء كبير من خلال توزيع الأحمال في أماكن مختلفة...، وهنا نأتي للنقطة المهمة، وهي أننا نحتاج إلى طريقة ما تأخذنا إلى أقرب Data center للمستخدم، وهذا يتم من خلال ما يسمى ب geoDNS، وال geoDNS هو اختصار ل Geographical Domain Name System، ويعني باختصار "عملية توزيع حركة المرور بناءً على location الخاص بال

Request"، هذا الأمر الجميل لديه مجموعة من التحديات التي يجب أن نأخذها بعين الاعتبار، وتتمثل ب:

a. Traffic redirection: تحتاج إلى أداة فعالة ومناسبة لتوجيه ال Traffic ووضعه في مكانه الصحيح، وال GeoDNS يمكنك استخدامه لهذا الغرض.
b. Data synchronization: يجب أن تضمن وجود نسخة احتياطية من ال cache أو database على أكثر من Data center أو يمكن لأكثر من data center الوصول لنفس البيانات التي تمت تعبئتها مسبقا من قبل مستخدمين في منطقة معينة، مثلا لو كان لديك Data center A و Data center B، وكلاهما نشطين، ثم صار Data center A بحالة offline، فيجب أن ينتقل ال Traffic إلى ال Data center B، ويجب أن يتمكن ال Data center B من الحصول على معلومات محدثة عن معلومات المستخدمين الذين تم نقلهم، وبكل تأكيد هذا يشمل لو أصبح الحمل 100% مثلا...
c. Test and deployment: من المهم جدا في حالة وجود أكثر من Data center أن تقوم بفحص النظام عندك في أكثر من Location، كما أنه من المهم بناء أو استخدام Automated deployment tools لتكون جميع ال Data center متسقة مع بعضها البعض!

إن وجود مثل هذه الأنظمة يتطلب منك عناية ونظرة شمولية حول ال service التي قمنا باستخدامها، لأن عملية التوسع تتطلب فصلا للعديد من المكونات لنتمكن من عمل scale بشكل مستقل لكل service، وهنا يظهر جليا دور ال Messaging Queue!

9. Messaging queue: تعد ال Messaging queue واحدة من أهم الاستراتيجيات المستخدمة والمطبقة في العديد من ال distributed systems، وال Messaging Queue يمثل نموذج asynchronous لحفظ المهام حتى يتم تنفيذها، والشكل ال Basic الخاص بها يتمثل بوجود ثلاثة أجزاء:
   a. Publishers أو ال Producers: أحد الأقسام الثلاثة والذي يمثل Client Application والذي سيكون دوره إنشاء لل Message التي يحتاجها، ثم يقوم بعمل Publish لها، في هذه العملية ستذهب ال Message وتوضع داخل Queue...
   b. Messaging Queue: الجزء الثاني هو ال Queue الذي يحتوي في ثناياه ال Messages التي تم استقبالها من ال Publishers لحين استدعائها ومعالجتها من خلال ال Subscribers...
   c. Subscribers أو Consumers: الجزء الثالثة هو ال Subscribers، ويمثل ال services أو ال server الذي سيقوم بالاتصال بهذه ال Queue ومن ثم القيام بعملية المعالجة التي يحتاجها بناء على ال Message المطلوبة...

   ملاحظة: هناك أشكال مختلفة من ال Messaging Queue، مثل:
   a. Point to Point: ببساطة سيتم إرسال ال Message هنا من ال Producers وسيتم استقبال الرسالة ومعالجتها من قبل Application واحد فقط (Only One Consumer)...

b. Publish/Subscribe: ببساطة يتم إرسال ال Message هنا من ال Producers ويمكن استقبال الرسالة ومعالجتها من واحد أو أكثر من ال Consumer، وهذا الفرق الجوهري عن النقطة السابقة!

Figure 1-17

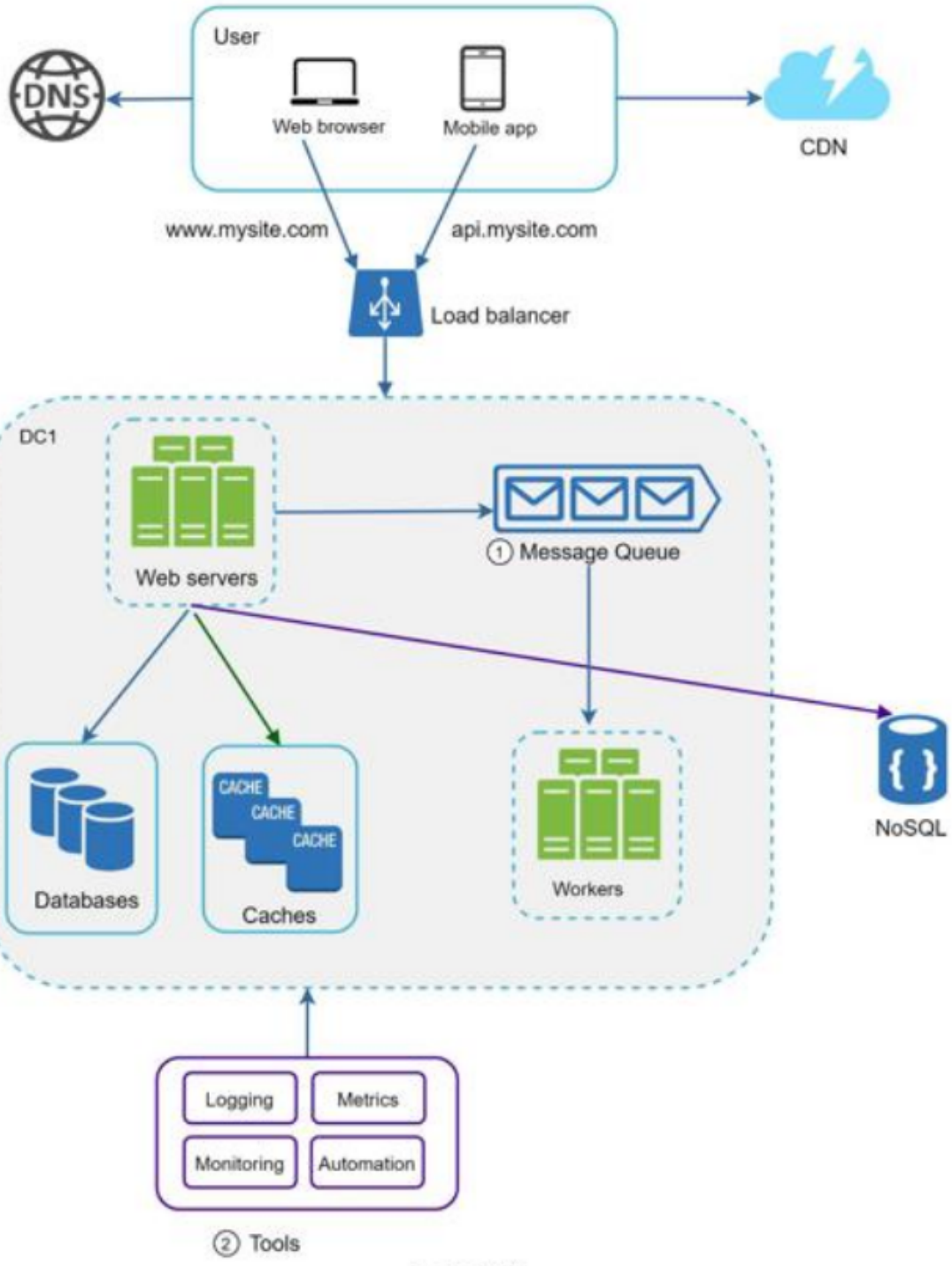
أعتقد أنك وجدت جمال هذه الاستراتيجية بعد هذه المقدمة البسيطة، فهي محببة للكثيرين؛ لأن ال Publisher لن يهتم لو كان ال Consumer متعطل أو غير موجود!، وكذلك ال Consumer لن يهتم لو كان ال Publisher متعطل أو غير موجود...، المهم أن يجد الرسالة داخل ال Queue، فلو افترضنا أن لدي تطبيق لمعالجة الصور، وقام ال Publisher بعمل Publish لعشرة صور، ثم صار offline، فإن ال Consumer عند اتصاله سيباشر العمل دون أن يبالي بال Publisher، أضف إلى ذلك أن معالجة الصور تحتاج عادة إلى وقت كبير وتحتاج إلى معالجة عالية...، لذلك، يمكنك بكل بساطة إنشاء Consumer حسب الحاجة، فلو احتجت Consumer 100 فيمكنك ذلك، وهذا يعني أنك قمت بعمل Scale ل service!، وبكل تأكيد إذا انتهت المهام يمكنك إغلاق ال Consumer التي لا تحتاجها...هذا الأسلوب باختصار يعطيك حرية ومساحة رائعة للعب مع المهام الموجودة داخل ال Queue، وأنت من يقرر كيف ستتعامل مع هذه المهام!

10. Logging, metrics, automation: من الأساليب والمهام التي نقوم بتطبيقها دوما مع التطبيقات أو الأنظمة التي نعمل عليها هي دمج أو بناء tools تساعدنا على دراسة وتتبع الأخطاء والحركات والمشاكل ونحوها على هذه الأنظمة، وتزداد أهمية هذا الأمر -بل لا يمكن الاستغناء عنه- عند وجود scalable system، لذلك يجب أن نأخذ بعين الاعتبار هذا الموضوع، ويجب أن نأخذ بعين الاعتبار تأثيره على قواعد البيانات!، لكن قبل ذلك، دعونا نلقي نظرة سريعة على هذه المفاهيم:
    a. Logging: ويقصد بها عملية مراقبة الأخطاء من خلال ال logs التي يتم تسجيلها عند حدوث أي مشكلة، وهي مفيدة جدا لتحديد سبب المشكلة ومعرفة مكانها، ومفيدة في معرفة حصول الخطأ أساسا!، ويمكن وضع ال logs على مستوى كل server على حدا، أو يمكن بناء أو استخدام tool لحفظ جميع ال logs بها...
    b. metrics: ويقصد بها مجموعة المقاييس التي يتم وضعها لمراقبة الحالة الخاصة بالنظام (هل يسير بشكله المعتاد، هل هناك تحسن، هل هناك مشكلة ما تسببت بفقدان traffic معين...الخ)، ويمكن القول أنها مراقبة "صحية" لحالة النظام، كما أن هذه المراقبة مفيدة جدا لتحقيق الرؤى الخاصة بال Business...، وهناك أكثر من مقياس يمكن استخدامه للحصول على فوائد متعددة في جوانب متعددة، نذكر منها:
        i. Host level metrics: وهي مراقبة على مستوى ال Host، مثل مراقبة ال CPU وال RAM.

ii. Aggregated level metrics: وهي مراقبة على مستوى مجموعة من العناصر أو الشروط للتحقق من أداء عنصر معين، مثل التحقق من ال Performance الخاص بال Cache tier أو database tier

iii. Key business metrics: وهي مراقبة على مستوى العناصر التي تضمن نجاح أو استمرار ال business، مثل نشاط أو عدد المستخدمين النشطين يوميا، فلو كان العدد يتراوح بين 1000 و 1100 ثم صار فجأة 400...سندرك أن هناك مشكلة ما يجب أن ننتبه إليها...

c. automation: ويقصد بها وسيلة التحكم والتطبيق أو التنفيذ أو المراقبة والفحص التي تتم بشكل آلي ودون تدخل بشري، وهذا أمر مهم جدا، فالأنظمة الكبيرة تحتاج إلى طريقة أو أداة تحسن من سرعة الإنتاجية وتقلل من نسبة الخطأ!، ومن الأساليب الجميلة والتي يمكنك استخدامها لهذا الغرض ال Continuous integration، ويقصد به مجموعة من الخطوات المتتالية التي يتم تنفيذها بناء على قواعد محددة تم وضعها مسبقا، فبمجرد حصول أي تغيير على الشيفرة البرمجية ورفعها يتم تنفيذ المهام مثل التحقق من القواعد التي كتبت فيها الشيفرة البرمجية والتحقق من ال unit test والقيام بعمل build من الشيفرة البرمجية الجديدة ونحو ذلك، وكل هذا يقبع تحت مظلة ال automation.


User
DNS
Web browser
Mobile app
CDN
www.mysite.com
api.mysite.com
Load balancer
DC1
Web servers
1 Message Queue
Databases
CACHE
CACHE
CACHE
Caches
Workers
NoSQL
Logging
Metrics
Monitoring
Automation
2 Tools

هل انتهينا هنا ^^؟ بالطبع لا، هناك أمر مهم يجب أن نأخذه بعين الاعتبار، وهو أن كمية البيانات التي تم الحديث عنها كمية كبيرة ومهولة، لذلك حان الوقت للحديث عن ال scale لل Data tier ^^

11. Database scaling: لقد تحدثنا سابقا عن ال Horizontal Scale وال Vertical Scale، وكنا قد تحدثنا حينها -النقطة رقم 4- عن ال Server scaling بشكل خاص، وحان الوقت الآن للحديث عن الموضوع الأخير في هذا الفصل، وهو ال Scaling لل Database، وهنا أيضا لدينا نفس التصنيف (Horizontal/Vertical):

a. Vertical Scale: ويطلق عليه أيضا scaling up، وهي عملية التمدد من خلال زيادة حجم المكونات ال Hardware مثل ال CPU وال RAM...إلى آخره، وعلى نفس الجهاز، فمثلا يمكنك أن تصل تقريبا لحجم 24TB من ال memory على instance واحدة من خلال أمازون!، وموقع مثل stackoverflow كان لديه قاعدة بيانات Master واحدة فقط حتى عام 2013!، ومع ذلك، فهذا الأمر له مساوئ متعددة، نذكر منها:

   i. حتى لو كنت تستطيع إضافة المزيد من المكونات وتحسينها، فهناك حد و limit لل hardware!
   ii. احتمالية الفشل كبيرة جدا، لأن لديك مصدر واحد فقط للمعلومة، فلو حصل أي عطل فجميع من يرتبط بك سيتعطل أيضا!
   iii. التكلفة العالية جدا، فسعر مثل هذه المكونات والعناية بها وتبريدها ونحو ذلك يحتاج مبالغ كبيرة...

b. Horizontal Scale: ويطلق عليه أيضا sharding، ويقصد به تجزئة قاعدة البيانات الكبيرة لقواعد بيانات أصغر لديها نفس ال Schema ووضعها على أكثر من سيرفر، كل قاعدة بيانات منها يسمى ب shard، قاعدة البيانات ال shard وإن كانت تشترك مع غيرها بال Schema، إلا أن البيانات التي تملكها فريدة لا تتكرر Unique، ويعمل هذا ال scale من خلال وضع function يمثل القاعدة التي سبتنى عليها ال shards، ولتبسيط الفكرة انظر للصورة:

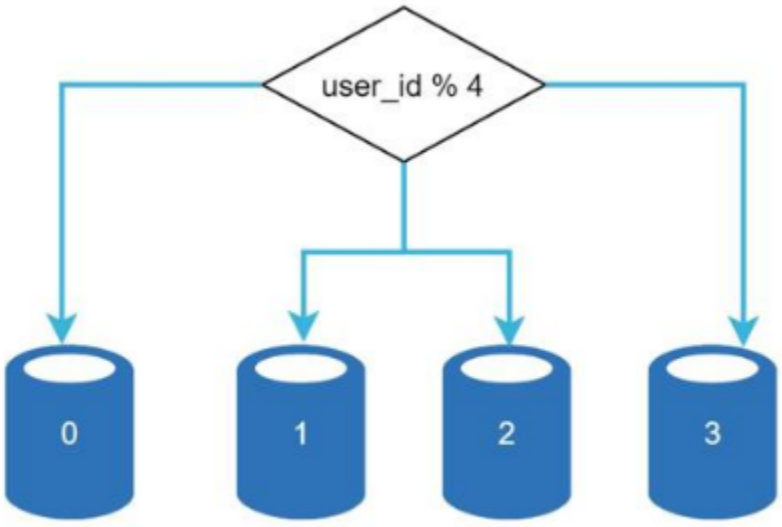


Figure 1-21

في هذه الصورة لدينا Hash function يقوم على حساب باقي القسمة لل user_id، فلو كان رقم المستخدم 0 فإنه سيكون في ال shard 0 وإذا كان رقم المستخدم 1 سيكون بال shard 1 وإذا كان رقم المستخدم 2 سيكون في ال shard 2 وإذا كان رقم المستخدم 3 سيكون في ال shard 3 وإذا كان رقم المستخدم 4 سيكون في ال shard 0...إلى آخره

شاهد هذه الصورة:

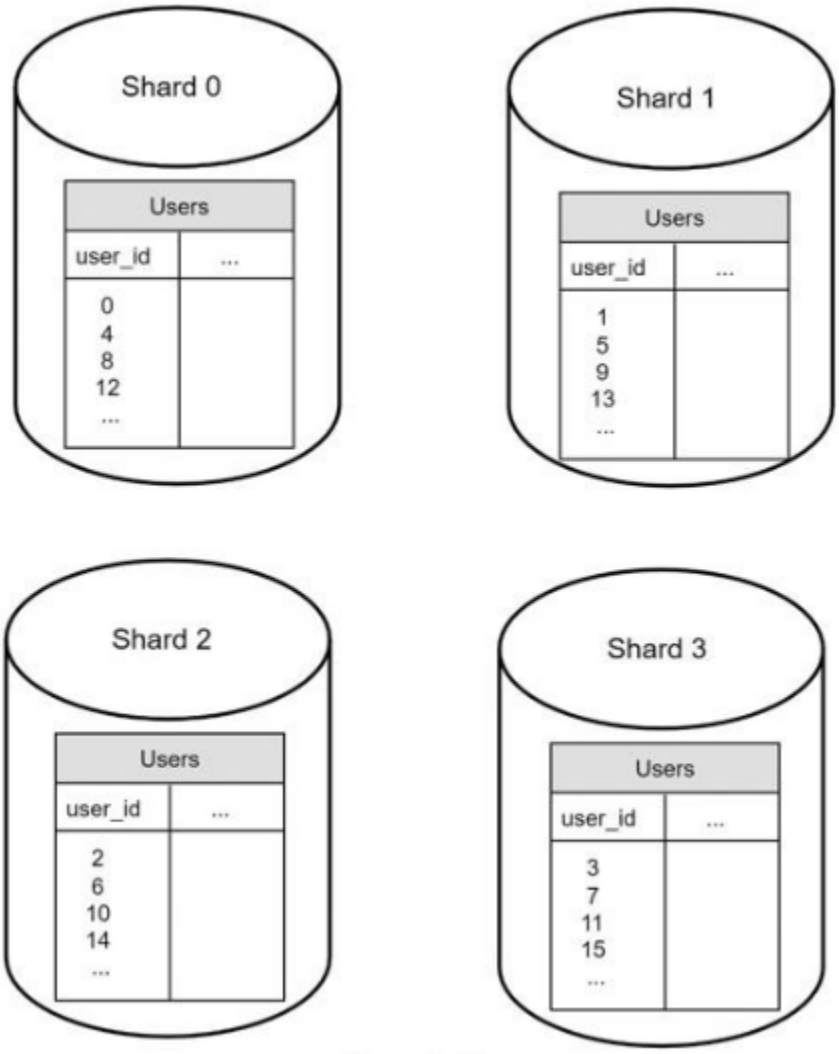


Figure 1-22

كما تلاحظ في المثال السابق، إن المفتاح الأساسي لنجاح ال sharding هو اختيار ال key المناسب، والذي يسمح لك بالوصول وتعديل البيانات بشكل سهل وسريع، هذا ال Key يمكن أن يكون عبارة عن Column واحد أو

أكثر، ويطلق عليه أيضا "Partition key"، وهذا كله جعل من ال sharding أسلوب جميل، لكن، لا تظن أن هذا الحل مثالي، فوجوده أدى لإنتاج تحديات وتعقيدات جديدة وجب أخذها بعين الاعتبار، منها:

A. Resharding data: من التحديات التي نتجت عن استخدام هذا الأسلوب هو الحاجة لوجود آلية أو طريقة لعمل Reshaeding للبيانات، وهذا قد يحدث أو يحصل لسببين أساسيين، الأول أن إحدى هذه ال shards قد تمتلئ سريعا ولا يمكنها أن تستقبل أو أن تتعامل مع بيانات جديدة، والثاني أن هذه ال shards قد تمتلئ بشكل غير منتظم فقد تجد ال shard 0 تحتوي على 50% من طاقتها الاستيعابية، بينما ال shard 1 ما زالت 6% وال shard 2 استهلكت 95%!، حدوث مثل هذه المشاكل يتم حله من خلال تحديث ال sharding function ومن ثم إعادة توزيع هذه البيانات للتناسب مع ال function الجديد...، واحدة من أشهر الطرق المستخدمة لحل هذه المشكلة: "Consistent hashing"، وفكرته ببساطة، قائمة على التأكد من أن عملية ال scale-up أو scale-down لن تجبرنا على تحديث جميع ال keys في جميع السيرفرات ولن تجبرنا على المرور على جميع السيرفرات، وإنما يكفينا معرفة عدد ال keys وعدد السيرفرات، وسيتم الحديث عن هذا لاحقا بإذن الله تعالى...

B. Celebrity problem: من اسم هذه المشكلة فهي تتعلق بالمشاهير فعلا ^^، ويقصد بذلك أن هناك حالات معينة من خلالها أو لأجلها يتم

زيادة ال load على shard معينة، فمثلا لو كان لدينا موقع فيه أعلام ومشاهير، أو منتجات معينة لحالات الطوارئ وحصل هناك خبر ما يخص أحد هؤلاء المشاهير أو كارثة تلزم شراء هذه المنتجات أو البحث عنها فإن ال load سيكون كبيرا، المشكلة هنا تكمن لو كانت هذه ال keys بنفس ال shard!!، سيصبح هذا ال shard 100% وغيره من ال shard ما زال مستريحا، هذه المشكلة يطلق عليها أيضا "hotspot key problem"، باختصار، يقصد بذلك تركيز ال traffices على إحدى ال shards دون غيرها، ويكون حل هذه المشكلة من خلال توزيع "المشاهير والأعلام" من الأشخاص أو المنتجات إلى shards مختلفة وعدم تجميعها أو منع تجمعها -قدر المستطاع- في shard واحدة، ويتم ذلك من خلال إعادة التقسيم لهذه ال shards...

ملاحظة: في الحالة الطبيعية عند وجود أشخاص مشاهير مثل إياد القنيبي وعبدالعزيز الطريفي وسامي عامري وإبراهيم السكران وأحمد السيد وغيرهم -حفظهم الله- في shard واحدة؛ فإن هذه المشكلة ستظهر، فوجودهم يعني traffic من المستخدمين لمتابعة أخبارهم وكتاباتهم ونحوها، وهذا بذاته سيجعلك تفكر منذ البداية بكيفية بناء ال shards أو إعادة توزيعها...

C. Join and de-normalization: من المشاكل المهمة لهذا الأسلوب هي صعوبة بناء جمل استعلام بين الجداول تربطها علاقة JOIN في حال وجود ال sharded على أكثر من server، ومن الحلول لهذه

المشكلة هو عمل de-normalization بحيث تنفذ كل جملة استعلام على جدول واحد فقط، هذه العملية حقيقة خطيرة ومهمة، ويجب عليك قبل تنفيذها أن تعرف حسنات وسيئات ال de-normalisation، ومتى يمكننا استخدامها، والعديد من الشركات الكبيرة تستخدم ال normalization وال denormalization معا...

الشكل النهائي لما قمنا بتصميمه حتى هذه اللحظة:

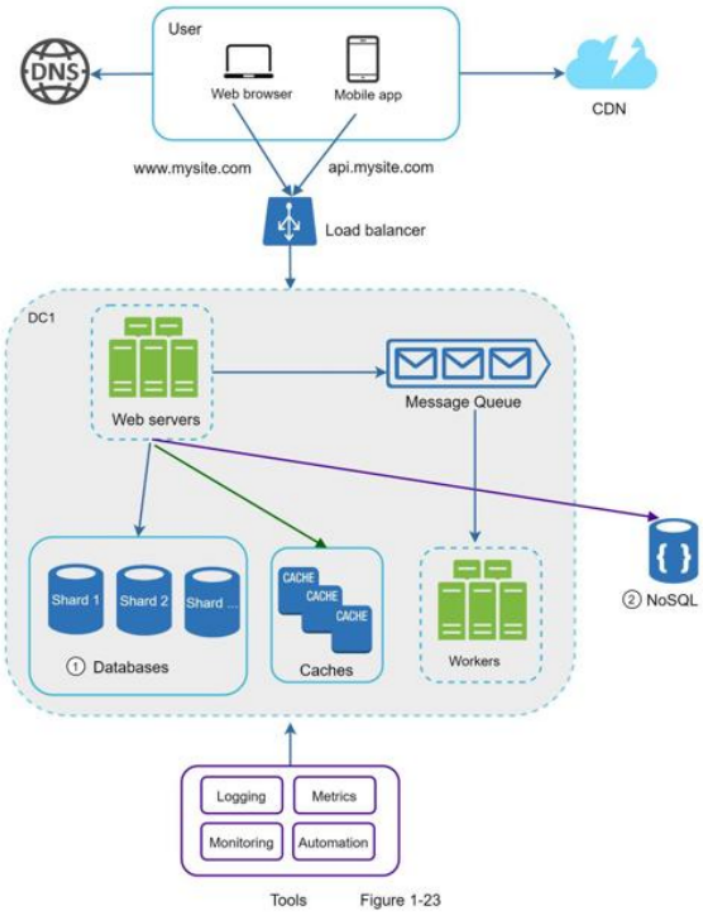


Figure 1-23

الخلاصة:

إن عملية بناء نظام متكامل يمكنه خدمة الملايين من المستخدمين موضوع كبير وفيه الكثير من الأساليب والتقنيات، وإن ما ذكرناه هنا لا يشكل إلا جزء من عدة أجزاء مستخدمة حول العالم، وهذه الأجزاء -الأساليب- يمكن أن تتغير أو تستخدم بطريقة مختلفة بشكل مختلف حسب حاجة كل نظام وكل مؤسسة، لذلك، يمكنك أن تعتبر هذه المواضيع وهذه الأبواب مجرد مقدمة لهذا الباب الكبير والواسع، لكن هذه المقدمة تغطي ما يمكن أن تفكر فيه وإن تنوعت الأساليب...

ونوجز ما تعلمناه أو تحدثنا عنه فيما يلي:

1. Keep web tier stateless
2. Build redundancy at every tier
3. Cache data as much as you can
4. Support multiple data centers
5. Host static assets in CDN
6. Scale your data tier by sharding
7. Split tiers into individual services
8. Monitor your system and use automation tools

فاصل معرفي NoSql UseCase

في حديثنا عن قواعد البيانات ندرك وجود أنواع منها، ال SQL Database وال NoSql Database، وهناك مجموعة من الأسباب التي جعلت من ال NoSql خيارا ممتازا لحل أو تجاوز بعض المشاكل الموجودة أو التي يمكن أن تنتج من استخدام قواعد بيانات علائقية، وهذه نقطة يجب أن تنتبه لها جيدا، فمعرفة أسباب استخدام النوع المناسب في الوقت والمكان المناسب، له أهمية بالغة في حياة المشروع وكيف سينعكس عليه ذلك، ومعرفة متى يمكنك الدمج بين النوعين سيضفي قوة إلى النظام الخاص بك...

الأسباب الشائعة لاستخدام ال NoSql (انتبه: ليس بالضرورة أن تكون الأسباب الشائعة سببا حقيقيا أو صحيحا يجعلك تختار هذه التقنية، وإنما هي بنيت على كثير مما يظنه المطورون حول ال NoSql أو بما يتناسب مع احتياجاتهم والمشاكل المتعلقة بأنظمتهم، وهذا دورك أيضا):

- Bigness: عند الحاجة إلى دعم البيانات الضخمة بمختلف أشكالها ووجود كمية بيانات ضخمة تحتاج إلى الفصل والتوزيع، فإن ذلك عادة يجعل الأنظار تتوجه إلى ال NoSql لتأدية هذا الغرض.
- Massive write performance: عند وجود حالات Write ضخمة، فإن الحاجة تكون ملحة عادة لاستخدام ال NoSql، ويعد هذا أحد أهم الأسباب الشائعة لاستخدام ال NoSql عند كثير من الناس...، فمثلا فيسبوك لديه مليار رسالة يوميا

من خلال ال Chat، فهنا يلزمك التفكير في طريقة للكتابة بسرعة عالية مع القدرة على توزيع وبناء cluster بما يتناسب مع عدد الرسائل هذا...

- Fast key-value access: الحاجة إلى الوصول السريع من خلال ذاكرة تخزين مؤقتة أطول من المتعارف عليه، لذلك يعد خيار ال NoSql خيارا مناسبا لهم إذا علمنا سرعة القراءة العالية اعتمادا على ال Hashed Key! (طبعا بعض أنواع ال NoSql تركز على الموثوقية، وبعضها بُني بطرق أخرى، فيجب أيضا أن يعي المطور ذلك...)
- Flexible schema and flexible datatypes: كثير من المطورين يفضلون استخدام ال NoSql لأن ال Schema وأنواع البيانات التي يمكن استخدامها وحفظها من خلالها كثيرة وبطريقة سهلة...
- Schema migration: في حالة ال NoSql يكون من السهل عمل Migration بدون خوف كبير، فهي مبنية أساسا بشكل Flexible، وهذا يعني القدرة على عرض البيانات بأكثر من شكل في أكثر من جزء في التطبيق...
- Easier maintainability, administration and operations: من السهل عمل scale والقيام بال operation ونحوها بتكاليف منخفضة
- Generally available parallel computing: وجود MapReduce من ضمن ال NoSql يعني القدرة على بناء parallel computing في المستقبل...

بعد تطرقنا لبعض الأسباب الشائعة لاستخدام ال NoSql، فيمكننا الحديث عن بعض الأسباب الأكثر تحديدا ودقة:

- إدارة ال Streams الضخمة لل non-transactional data مثل ال Application Log...
- سرعة استجابة سريعة حتى بوجود أحمال (loads) عالية
- يتجنب ال JOIN المعقدة الموجودة بال RDBMS خصوصا عندما تصبح كبيرة الحجم
- مفيدة في حالة ال Soft real-time systems عندما تكون ال low latency مهمة.

ملاحظة: يقصد بال Real-time system هي تلك الأنظمة التي يشكل ال deadline لها نقطة فاصلة أو نقطة مهمة في سيرها، وكل job في هذه الأنظمة لديها deadline تنتهي فيه، إذا حدث أي خطأ فقد يتسبب ذلك بمشاكل كبيرة، ويمكن تقسيم هذه الأنظمة لجزئين الأول: Hard real-time system والثاني Soft real-time system، ويقصد بال Hard real-time system هي تلك الأنظمة التي ستعطي نتيجة خاطئة -وإن كانت القيمة صحيحة- إذا لم تقدم في الوقت المحدد!، ومثل ذلك الأجهزة الطبية أو وحدات تحكم المطارات ونحو ذلك، مثلا؛ لا يمكن أن نقبل التأخر لمدة ثانية لمعرفة نسبة الأوكسجين، فحتى لو كانت القيمة صحيحة، إلا أن هذا التأخر قد يؤذي المريض، لذلك فالوقت هنا مهم جدا ويهتم بأجزاء الثانية، أما النوع الثاني فهو ال Soft real-time system، وهي تلك الأنظمة التي تتراجع أهمية نتائجها كلما تأخر الوقت، وهذا قد يتسبب بخسائر أو مشاكل كبيرة للنظام، فتصبح النتائج بدون قيمة كما يرجو المستخدم، وهذا مثل الألعاب الرقمية، فالوقت مهم للاعبين أثناء اللعبة...، لذلك قد تجد هنا لاعبين اثنين بسرعة استجابة مختلفة يلعبان نفس اللعبة لكن يتميز أحدهما عن الآخر بسرعة الحركة أو الاستجابة، وهذا ينعكس إيجابا على أدائه في اللعبة...

- الإضافة والحذف والتعديل والاستعلام في الوقت الفعلي (real time)
- Real-time page view counters

هناك الكثير من الأسباب الشائعة وغير الشائعة التي قد تدعوك لاستخدام ال NoSql، وإن كان ما أوردناه هنا موجزا إلا أنه سيعطيك لمحة مهمة عما تحتاج إلى فعله عند نظرك إلى هذه التقنية أو إلى الحوارات التقنية التي تدور في هذا الفلك...

فائدة

إن الفكر النسوي تغلغل إلى الأعماق حتى صرت تجده عند الرجال والنساء، وهذا أمر خطير، فكثير ما ترى أب يمنع ابنته من الزواج حتى تنهي تعليمها الجامعي وتبدأ بالعمل حتى لا يتحكم بها زوجها!، وفي زوج يعير زوجته بإنفاقه عليها، وفي أم تهيء ابنتها لبيت الزوجية بكلمات التخويف والحذر والسيطرة...، وآخر يُعير لأنه أب للبنات!...، إن هذه الأفعال كلها لا تخرج إلا من منظومة فكرية فاسدة تغلغلت في أعماق المجتمع!

BACK-OF-THE-ENVELOPE ESTIMATION

في هذا الموضوع سنتحدث عن جزئية مهمة ستغير من طريقة تفكيرك في التعامل مع الأسئلة أو المشاكل التي تتطلب حلولا سريعة ودون الحاجة إلى أن تكون الحلول دقيقة 100%!

إن الجزء الأول الخاص بالعنوان يعني ظهر المغلف، وهي إشارة تعني إلى الحاجة إلى حساب أمر ما بسرعة، ثم يأتي الجزء الثاني ليخبرك بأن الكتابة على ظهر هذا المغلف كانت للحصول على نتيجة تقديرية، وبهذا فيكون الهدف من هذا الموضوع هو الوصول إلى نتيجة أفضل من مجرد التخمين وأقل من أن تكون النتائج صحيحة 100%!، بل سيكون هدفك أن تقدر نتائج قريبة من الحقيقة أو تعطيك تصورا قريبا عن الحقيقة...

والآن حتى تستطيع البدء، هناك عدة مفاهيم يجب أن تدركها أو أن تتذكرها وهي:

- Power of two: مع أن هذا المفهوم سهل وبسيط، وقد تعلمناه في مراحل الدراسة المبكرة، إلا أنه يعد واحد من أهم الأساسيات التي يمكنك أن تنطلق منها لحساب العمليات المعقدة أو الكبيرة...، ويرتبط هذا المفهوم بشكل جلي في علوم الحاسب الآلي، فكلنا يعلم أن الحاسب الآلي ما هو إلا 0 و 1، وهذا يتمثل في حجم البيانات، فنحن ندرك أن ال Byte هو 8bits، وأن الرقم 2 للقوة 10 = 1024 وأن 2 للقوة 20 = مليون، وأن 2 للقوة 30 = مليار!

  لحظة، لحظة، أعلم أن لسان حالك الآن، لا، هذا ليس صحيحا، ف 2 للقوة 20 =

1048576!، لكن، هل هذا الأمر مهم في هذا السرد؟!، بكل تأكيد لا، إننا نتحدث ونطبق المبدأ الذي تكلمنا عليه، جميعنا يعلم حفظا أن 2 للقوة 10 هي 1024، لكن كلما تضاعفت القيم كلما كان حساب ذلك أكثر صعوبة!، فبدلا من أن أضيع وقتي ووقت من السائل أو المقابل، يكفيك بالإجابة التقديرية عن هذا السؤال...والآن لو أخبرتك في مقابلة ما، ما هي نتيجة حساب 2 للقوة 40، فجوابك الكافي هو 1TB!، ولا داعي لتجلس في الساعات تحسب ذلك على ظهر المغلف!، فظهر المغلف قد لا يتسع لمثل هذه العملية، وإن وسع ذلك فإنك خسرت من وقتك الثمين كثيرا!، أضف إلى ذلك، أن الوحدات المستخدمة لقياس أحجام البيانات مهمة جدا مثل ال bit, byte, kilobyte, megabyte, gigabyte, terabyte, petabyte، وفائدتها الجميلة هي إعطائك الفرصة والقدرة على التحويل من حجم لحجم أو تقدير إجابة معينة من خلال فهمك لهذه الوحدات والفروق التي بينها...

- Rule of 72: هذه قاعدة بسيطة تسهل عليك حساب سؤال مهم، متى يمكن أن تتضاعف ثروتك؟ ومتى يمكن أن يتضاعف حجم البيانات عندك؟ ومتى يمكن أن يتضاعف أعداد المستخدمين النشيطين لديك؟...إلى آخره، والجواب بكل بساطة يكمن في قسمة الرقم 72 على النسبة المئوية للزيادة، وبهذا تكون النتيجة الرقم التقديري حتى تتضاعف القيم لديك...مثال، لو أن الاستثمار الخاص بك يعود عليك بدخل وزيادة سنوية بمقدار 10% دائما، فإن الوقت المقدر لتضاعف القيمة هو 72 على 10 ليكون الناتج هو 7 سنوات تقريبا!
- Availability numbers: ويقصد بهذا المفهوم قدرة النظام على الاستمرار بالعمل لأطول مدة زمنية ودون انقطاع بالخدمات، ويتم الحديث عن هذه الأرقام وتمثيلها

من خلال النسبة المئوية مثل 100% والتي تعني أن النظام لن يسقط (Zero Downtime)، وهذا تجده موثقا في الأنظمة للشركات الكبيرة ضمن اتفاقية ال SLA، وال SLA هي اختصار service level agreement، والتي بدورها تمثل اتفاقية بين الزبون ومقدم الخدمة لبيان مدة استمرارية عمل النظام دون توقف، فمثلا ستجد المواقع تضع ال SLA uptime لديها يساوي أو أكبر من 99.9%، وقد تظن لوهلة أن 99.9% أو 99.99% أو 99.999% هي أرقام للغايات التسويقية فقط، لكن في الحقيقة هناك قيمة مهمة تختفي خلف هذه الأرقام، شاهد هذا المثال:

لو افترضنا أن Availability كانت 99%، فإن أقصى مدة زمينة لل Downtime يجب أن لا تتجاوز ال 14.4 دقيقة يوميا، وفيما يعادل 3.65 يوم في السنة، ويمكن حسابها ببساطة من خلال هذه المعادلة -باعتبار أن الزمن المطلوب بالدقائق-:

MinutesInDay - MinutesInDay * Availability = downtime

وهذا يعني: 1440 - 1440 * 99% = 14.4 دقيقة يومي (14.4 * 365 = 3.65 يوم سنوي)، الآن، لنشاهد الفرق بوجود تسعات أكثر بعد الفاصلة، وأنها ليست مجرد عملية تسويقية!، فلو افترضنا أن Availability كانت 99.99% فالنتيجة ستكون 1440 - 1440 * 99.99% = 0.144 (وهذا يعادل 8.64 ثانية يوميا)، وسنويا (0.144 * 365 = 52.56 دقيقة)...، لاحظ الفرق بين النتائج، انخفض أقصى وقت لل downtime من 3.65 يوم إلى 52.56 دقيقة!، لذلك، احرص على تذكر هذا في الأنظمة التي تعمل عليها...

بعد حديثنا عن هذه النقاط، يمكننا أن ننطلق إلى مثال تطبيقي ومهم لتقدير ال Requirements الخاصة بنظام ستكون مسؤولا عن بنائه...

مثال: قدر المتطلبات اللازمة لحساب ال QPS وال Storage لموقع تويتر إذا علمت أن (الأرقام ليست حقيقة، وإنما لأغراض السؤال):

- هناك 300 مليون مستخدم شهري نشط للموقع
- 50% من المستخدمين النشطين يزورون الموقع يوميا
- كل مستخدم يقوم بنشر تغريدتين يوميا بالمتوسط
- 10% من هذه التغريدات فيها Media
- البيانات التي تم رفعها ستحفظ لمدة 5 سنوات كحد أقصى

ملاحظة: QPS يقصد بها Query Per Second، ويقصد ب DAU المستخدمين النشطين يوميا(Daily Active User)

التقدير (من هنا نبدأ بالحل):

- الجزء الأول، تقدير ال QPS، ويكون كالتالي:
  - المستخدمين النشطين يوميا = 300 مليون * 50% = **150 مليون**
  - التغريدات QPS = المستخدمين النشطين يوميا 150 مليون * 2 (تغريدتين يوميا) / 24 ساعة / 3600 ثانية = 3500 QPS(تقريبا)
  - أكبر أو أعلى QPS يمكن أن يحصل (Peak QPS) = التغريدات QPS وهي 3500 * 2 (تغريدتين يوميا) = 7000 QPS تقريبا.

ملخص النقاط التي بالأعلى: لتقدير عدد ال Query التي يمكن أن يقوم المستخدمين بتنفيذها بالثانية يساوي عدد النشيطين منهم يوميا مضروبا بمتوسط عدد التغريدات لديهم، لكن هذا لا يعد كافيا، فهناك أوقات قد يصل فيها المستخدمين إلى ذروة التفاعل مثل دخول المستخدمين بنسبة 100% إلى النظام وهذا يعني أننا ملزمين بأخذ ال Peak بعين الاعتبار وأن نتعامل على أساسه أو على أنه موجود لتكون متطلبات النظام قابلة للتمدد لتغطية هذه القمة...

- الجزء الثاني، **تقدير ال Storage لل Media فقط**:
  - متوسط حجم التغريدة (افترض أرقاما بناء على المعطيات التي عندك، فإن لم تكن موجودة يمكنك أن تفترض قيما وتكمل الحل بناء عليها، وهذا مفيد في الإجابة على هذا النوع من الأسئلة في المقابلات، وهنا قمنا بافتراض أن التغريدة لديها id, text, media):
    - tweet_id: 64 bytes
    - text: 140 bytes
    - **media: 1MB**
  - بناء على ما افترضنا سابقا لمتوسط حجم التغريدة فإن
    مساحة التخزين (Media Storage) = المستخدمين النشطين يوميا 150 مليون * تغريدتين * 10% (وهي متوسط عدد التغريدات التي تحتوي صورا) * 1MB حجم ال media التي فرضناها = 30TB يوميا!
  - سيتم حفظ ال Media لمدة 5 سنوات فهذا يعني 30 تيرابايت * 365 * 5 = 55PB.

ملخص النقاط التي بالأعلى أننا نحتاج حساب المساحة التخزينية المطلوبة بشكل يومي للمستخدمين النشطين، لكن حتى يتم هذا يجب أن نأخذ متوسط التغريدات التي يتم مشاركة Media فيها بعين الاعتبار، وحسب المثال الخاص بنا كانت 10%، بعد ذلك نأخذ أقصى حجم / متوسط الحجم لل Media والذي افترضنا ب 1MB ليظهر لدينا الحجم اليومي من ال media المطلوب حفظها...ثم سنأخذ بعين الاعتبار أن هذه ال Media ستبقى لمدة 5 سنوات، وبهذا نكون قد قدرنا ال Storage المطلوبة وعدد ال Query التي سيتم تنفيذها بالثانية...

بناء على ما سبق، نستنتج أن عملية التقدير تعطيك نظرة قريبة لما سيكون عليه النظام الخاص بك، وهي بكل تأكيد أفضل من مجرد التخمين الذي يضيع وقتك وجهدك بدون مرجعية حقيقية...

وإليك بعض النصائح المفيدة في عمليات التقدير التي يمكن أن تستخدمها في حياتك العملية والمهنية، والمفيدة لك أثناء المقابلات:

- دوما فكر في التقريب إذا كانت النتائج الدقيقة غير مهمة، مثلا لو سألك أحدهم في مقابلة ما ناتج هذه العملية 99987 / 9.1 فيكفيك أن تقوم بتقريبها لتصبح 100000 / 10!، بكل بساطة إضاعة الوقت لحاسب الأعشار هنا عملية غير مهمة، وكذلك لو سألتك ما ناتج ضرب 99 * 99، ببساطة يمكنك جعلها 100 * 99 ثم طرح 99 من النتيجة...!، ما جعل الأمر سهلا هنا هو عملية تقريب أو تدوير المعطيات لنصل إلى نتيجة قريبة من الصحة دون إهدار الوقت ودون تخمين...

- إذا قمت بفرض معطيات معينة كما فعلنا في المثال الخاص بتويتر، فقم بتوثيق الافتراضات التي قمت بكتابتها للرجوع إليها عند الحاجة، وليدرك من يقرأ تلك الأرقام مقصدك
- اكتب الوحدات بجانب الأرقام الخاصة بك...، مثلا لا تقم بكتابة 5، بل اكتب 5 MB، ولا تقم بكتابة 1 بل اكتب 1 seconds...وهكذا
- يمكنك التدرب على حساب QPS, peak QPS, storage, cache, number of servers...إلى آخره، وهناك أمثلة كثيرة يمكنك محاولة حلها أو افتراضها.

فائدة

إن أي مشروع دعوي لا يقاس بعدد الأتباع والمتبوعين!، بل يقاس بالحق الذي فيه، ولا تقاس كفاءة الأتباع والمتبوعين فقط بالإنضمام أو بالتعداد، بل تقاس بمقدار تبصر الأتباع والمتبوعين في الحق الذي هم عليه!

A FRAMEWORK FOR SYSTEM DESIGN INTERVIEWS

في هذا الفصل من الكتاب يقوم Alex بالتركيز على جانب مهم في المقابلات التقنية والتي تتعلق بال System Design، وقد قام بتلخيص أهم الأفكار التي ينبغي عليك إدراكها وفعلها، وأهم الأشياء التي ينبغي عليك تجنبها، ونوجز ذلك على شكل (موجز، افعل، ولا تفعل)...

1. الموجز: اعلم أن مقابلات ال System Design وضعت لاستكشاف مفاتيح الكنوز التي لديك، فلا يمكن للعقل أن يتصور أن هناك من يستطيع تصميم بنية تحتية لشركة مثل جوجل أو فيسبوك أو تويتر في جلسة لمدة ساعة!، وإنما المراد هو البحث واستكشاف ما لديك من أفكار وحلول، والقدرة على التواصل والعمل مع الفريق وتحمل الضغط ونحو ذلك...، هناك مجموعة من الخطوط الحمراء التي ينبغي عليك تجنبها مثل العناد وحصر الأفق الخاص بك في جانب واحد من المشكلة ونحو ذلك، وأكبر مصيبة يمكن أن تقع بها هي ال over-engineer، فعليك دوما بناء المطلوب، فكثير من المشاكل التي تتطلب تكاليف باهظة هي نتيجة أخطاء من هذا النوع، لذلك احرص على فهم المشكلة بشكل صحيح، اسأل عما ينقصك من تفاصيل، ناقش المقابل لك، شاركه اقتراحاتك، اقبل توصياته ثم تابع...، ويمكن تقسيم الأجزاء التي ينبغي لك الاهتمام بها إلى:
    a. افهم المشكلة جيدا وحدد النطاق الخاص بالمشكلة

b. قدم تصميم High-level وحاول أن تحصل على الدعم من خلال جعل العملية تشاركية بينك وبين من يقابلك
c. قم بالدخول بشكل أعمق فيما يخص ال Component، وتلخيص الخرابيش التي قمت بكتابتها في الخطوة السابقة، خذ أفكارا من المقابل لك، واطرح أفكارا بناء على التغذية الراجعة...
d. قم بتلخيص ما قمتم به في هذه المقابلة وتحدث مع المقابل، إذا سألك عن المشاكل التي تتوقعها في ال System Design الخاص بك فاحرص على أن تجيبه بصدق، فلا يوجد نظام خالي من الأخطاء، أنظر إلى المشاكل المهمة وحاول طرح حلول لها...

2. افعل:

a. اسأل عما ينقصك من المعلومات ولا تفترض معلومات من عندك وتعتبر أنها صحيحة، فمثلا قبل أن تفترض عدد المستخدمين النشطين في الموقع الخاص بك، اسأل عن هذا العدد، فإن أخبرك المقابل بعدد؛ فاستعمله، وإلا فافترض رقما من عندك وقم بتوضيحه في المثال الخاص بك...
b. افهم المطلوب منك جيدا، وافهم متطلبات النظام الإلزامية...
c. الحل الخاص بك ليس بالضرورة أن يكون أفضل أحل، وليس بالضرورة أن يكون صحيحا، والحل الصحيح عند التعامل مع الشركات الناشئة قد يكون خاطئا عند التعامل مع الشركات الكبيرة...لذلك افهم هذه الاختلافات جيدا وتعامل مع ال System Design والمقابل على هذا الأساس...

d. فكر بصوت مرتفع، وأقصد بذلك شارك الأفكار وتواصل مع المقابل لك حول الأفكار التي تدور في ذهنك وتنوي المباشرة بها...، أي اجعل هناك حلقة من التواصل بينك وبينه، لكن لا تبالغ!
e. حاول طرح أكثر من حل للمشكلة إذا كان هذا ممكنا
f. إذا تم الاتفاق على النسخة الأولية للتصميم، فباشر بكتابة / رسم التصميم من أعلى مستوى وليس من الأسفل!، فليس من المعقول أن تذهب وتكتب تفاصيل جزئية ما، ولم ترسم المكونات التي ستتعامل معها...!
g. إذا لم تستطع أن تجد حلا أو وجدت نفسك عالقا ولا تستطيع إيجاد الحل المناسب أو المتابعة، فلا تستلم، حاول طرح بعض الأفكار أو الأسئلة أو اطلب المساعدة من المقابل...

3. لا تفعل:

a. إياك أن تتعجل الحل، أعطي نفسك فرصة لفهم المطلوب، اسأل عما ينقصك، ثم باشر بالحل...
b. إياك أن تأتي مثل هذه المقابلات ولم تقم بالتحضير لها...
c. لا تقم بكتابة الكثير والكثير من التفاصيل لل Component الخاصة بك في أول الأمر أو عند ال High level design، ادخل إلى التفاصيل تدريجيا خطوة خطوة بالتسلسل...يعني من ال High level إلى ال Deep level.
d. إياك أن تظن أن المقابلة انتهت بمجرد تقديمك لل Design، فهناك أمور تبدأ من هذه المرحلة، فكن مستعدا لمتابعة المقابلة ما لم يخبرك بالمقابل بأن المقابلة قد انتهت ^^

فائدة

لا ينبغي للمسلم أن يظن أنه ناج من العذاب لكونه مسلما دون أن يأخذ بأسباب النجاة!، إن الله -سبحانه وتعالى- رحمن رحيم، نحبه ونحسن الظن به، وحسن الظن به يعني أن نكون كما أمرنا، وهو الرحمن الرحيم!، لذلك، تجد كلُّ من بَشَّرَهُ رسولُ الله -صلى الله عليه وسلم- بالجنة أو أخبره بأنَّه مغفورٌ له؛ لم يَفْهَمْ منه هو ولا غيرُه من الصحابة إطلاق الذنوب والمعاصي له ومُسامَحَتَهُ بترك الواجبات، بل كان هؤلاء أشدَّ اجتهادًا وحذرًا وخوفًا بعد البشارة منهم قبلَها؛ كالعشرةِ المشهودِ لهم بالجنة، وقد كان الصدِّيقُ شديد الحذَر والمخافةِ، وكذلك عمرُ؛ فإنَّهم علموا أن البشارة المطلقة مقيَّدةٌ بشروطها والاستمرار عليها إلى الموت، ومقيَّدةٌ بانتفاءِ موانِعها، ولم يَفهمْ أحدٌ منهم من ذلك الإطلاقَ والإذنَ فيما شاؤوا من الأعمال...، وهذه رسالة مهمة لكل من تعلق بالأماني وترك والعمل!

DESIGN A RATE LIMITER

ال Rate limiter أحد الطرق الجميلة للتحكم في أقصى عدد مسموح به من ال traffic لمستخدم ما أو خدمة ما بناء على شرط معين، ومن الأمثلة -في عالم ال HTTP- التي سمعنا عنها بكثرة أو رأينها بكثرة تحديد عدد ال API Request المسموح للمستخدم في إرسالها خلال دقيقة، أو إنشاء منشورين كحد أقصى للمستخدم في الثانية...إلى آخره، فإذا جاء ال client أو ال service وتجاوز الحد، فسيتم عمل Block له مباشرة، وعادة ما يتم إرجاع ال Status رقم 429 ك Error Status Code في هذه الحالة، والتي تشير إلى Too Many Request، ويجب أن تراعي عند استخدامك لهذا الأسلوب أن تحافظ على معدل الاستجابة السريعة لل Response، واستهلاك أقل قدر ممكن من ال Memory، وأن لا يتعطل ال System في حال تعطل ال Rate Limiter أو وجود مشكلة ما فيه...إلى آخره

ولهذا الأسلوب العديد من المزايا الجميلة، وهي:

1. يساعد على الحماية من هجمات ال DDOS
2. تقليل التكلفة، وخصوصا عند وجود Third party، فكثير من المكاتب ستحاسبك على كل Api request، وإن كنت تظن أن سعر التكلفة القليل لن يؤثر عليك، فكر بنتائج هذه القيم: مثلا 0.001 * 10000Request, ثم تخيل 100000 وتخيل 1000000...إلى آخره، واعلم أن الوصول إلى هذه الأرقام سهل وليس بالصعب...

3. يقلل من احتمالية ال overload لل server وذلك من خلال التمييز بين ال Request والحفاظ على ال server من الأخطاء الغير مقصودة للمستخدمين...

والآن بعد هذه المقدمة البسيطة، لننتقل إلى جزئية أخرى، وهي تصميم ال Rate Limiter ^^، وبكل تأكيد هناك عدة طرق لتنفيذ ووضع حد لل Traffic، وهذا الأمر يجب عليك أن تدركه، وسيقودك حتما إلى وضع مجموعة من الأسئلة لبناء التصميم المناسب لل Rate Limiter، منها:

- هل سيتم تطبيق ال Rate Limiter على Client-Side أم على ال Server-Side؟
  ملاحظة: التنفيذ عادة ما يكون على مستوى ال Server-Side لأنه لا يمكنك الوثوق بال Client، فهناك احتمالية عالية دوما لتجاوز ال Client، كما أنك قد تفقد السيطرة أو قد لا تمتلك السيطرة الكاملة على التحكم بما يحدث من تنفيذ للأوامر عند ال client، وطبعا ليس بالضرورة أن يكون ال Rate Limiter على مستوى ال server نفسه، بل قد يكون ك Middleware قبل الوصول لل Api server...

- سيتم وضع القيود والشروط لل API requests بناء على ماذا؟ على ال userId أم ال IP address...إلى آخره، وهل يمكن أن يكون هناك أكثر من شرط أو مجموعة لهذه القيود -قيود مرنة-؟

- هل سيتعامل النظام مع كمية ضخمة من ال Requests أم كمية قليلة؟ (لشركة كبيرة أو حاضرة أم لشركة ناشئة؟)

- تشغيل ال system سيكون على distributed environment؟

- ال Rate Limiter سيكون Service منفصلة أم سيكون موجودا على مستوى ال Application code؟

- هل سيتم إشعار المستخدمين إذا تجاوز أحدهم ال Rate Limit؟

فاصل معرفي:

ال Microservice: يترجم هذا المفهوم إلى الخدمات المصغرة، وهو عبارة عن نهج معماري من نوع cloud native؛ يتكون فيه التطبيق الواحد من العديد من المكونات أو الخدمات الصغيرة المقترنة فيما بينها، كل Service تقدم خدمة محددة منفصلة أو مرتبطة بمكونات أخرى بشكل غير وثيق وقابلة للاستخدام وال Deploy باعتبارها كينونة مصغرة ومستقلة...

ال Cloud Native: هو مفهوم يشير إلى كيف يمكننا إنشاء ال Service وكيف يمكننا عمل ال Deploy لها أكثر من الاهتمام أين سيتواجد التطبيق...

ال API gateway: هو مفهوم يشير إلى software يقدم عادة مجموعة كبيرة من الخدمات التي يتم تنفيذها بناء على الـ Request القادمة من المستخدمين، ويتم توجيهها إلى Service أو أكثر من Service على ال Backend، ويقوم بجمع البيانات وإرسالها للمستخدم عند جاهزيتها، مجمعة أو غير مجمعة بما تقتضيه حاجة العمل، ومن الخدمات التي تقدمها rate limiting, SSL termination, authentication, IP, whitelisting, servicing static content، data analytics and security layer، باختصار هي Middleware تقدم لك خدمات متعددة منها ال rate limit...

Algorithms for rate limiting

هناك العديد من الخوارزميات التي يمكنك استخدامها لتطبيق ال Rate Limit، كل واحدة من هذه الخوارزميات لها حسناتها وسيئاتها، وما يلزمك هنا هو القدرة على اختيار الخوارزمية الأفضل حسب المعطيات التي أنت بحاجة إليها...وأشهر هذه الخوارزميات هي:

- Token bucket: تعد هذه الخوارزمية واحدة من أشهر الخوارزميات ومن أكثرها انتشارا بين المطورين، وذلك لسهولة استخدامها وفهمها وبساطتها، كما أنها تستخدم في العديد من الشركات التقنية مثل أمازون!، ومبدأ عمل هذه الخوارزمية قائم على بناء Container فيه عدد من ال Token التي تم تعريفها مسبقا، إذا كان حجم ال Container لديه القدرة على احتمال أربعة من ال tokens، فإن أي tokens يزيد عن ذلك سيتم طرحه خارجا، كل API request سيقوم بسحب واحدة من ال tokens، عملية ال refill (إعادة التعبئة) تتم بناء على شرط معين، مثل يتم إضافة

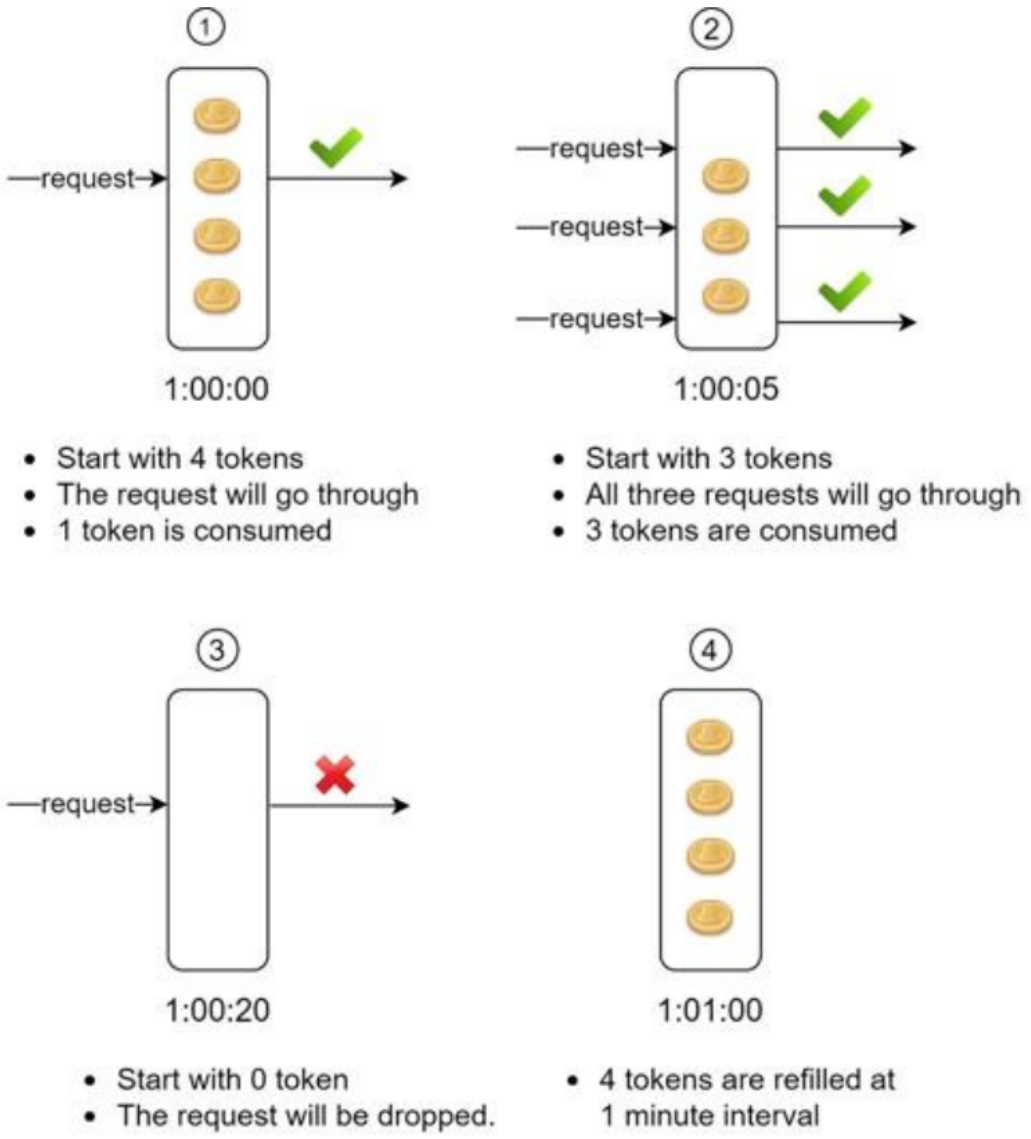
Two tokens في كل دقيقة، إذا انتهت ال tokens قبل انتهاء الدقيقة وجاء request جديد، فهذا ال request لن يتم التعامل معه وسيتم إسقاطه...، وبناء على ما سبق، يمكننا القول أن هذه الخوارزمية تأخذ Two Parameter، الأول يحدد عدد ال Tokens في ال Container، ويطلق عليه Bucket Size، والثاني المسؤول عن تحديد عدد ال Tokens التي سيتم إضافتها إلى ال Container في كل ثانية، ويطلق عليه Refill Rate...

وهذا يقودنا لجزئية أخرى مهمة، وهي كم Bucket نحتاج أساسا في المشروع الخاص بنا؟، والإجابة على هذا السؤال تكون من خلالك أنت!، لأنك من يستطيع تحديد الشروط الخاصة بال Rate Limit، لكن، خذ الأمثلة ليسهل عليك الأمر...

إذا كان لديك موقع تواصل اجتماعي ولديك شروط تمنع إنشاء أكثر من منشور واحد للمستخدم في الثانية، وتمنعه من إضافة أكثر من 100 صديق في اليوم، وتمنعه من إبداء إعجابه أكثر من 5 مرات في الثانية، فأنت بحاجة إلى ثلاث Bucket لكل مستخدم، ولو افترضنا أنك ترغب بتحديد عدد ال Request على مستوى ال IP address، فيلزمك bucket لكل ip address، ولو افترضنا أنك ستسمح ب 10000 Request بالثانية كحد أعلى، فأنت بحاجة إلى Bucket واحدة Globally يتم مشاركة كل Request معها...

بعد هذا كله، هناك نقطة سيئة تخص هذه الخوارزمية، وهي صعوبة ضبط ال Refill Rate وال Bucket Size، لذلك يجب أن تكون حريصا ومدركا للقيم التي ستقوم بتحديدها هنا...

شاهد الصورة والتي تمثل الفكرة الخاصة في هذه الخوارزمية:



- Leaking bucket: هذه الخوارزمية شبيهة بخوارزمية Token bucket، إلا أنها قوم بحفظ ال Requests داخل Queue، وغالبا ما تكون هذه ال Queue من نوع FIFO، ما يتم إضافته أولا سيتم استدعائه أولا...، وطريقة عمل هذه الخوارزمية هي:

يأتي ال Request إلى ال Queue، إذا كانت ال Queue ممتلئة فسيتم عمل Drop لهذا ال Request، وإذا كان هناك حيز متاح فسيتم إضافة هذا ال Request إلى آخر ال Queue، ويتم سحب أول عنصر تم إضافته ثم الثاني ثم الثالث وهكذا بالترتيب...، ولهذه الخوارزمية Two Parameter أيضا، الأول هو Bucket size وهو مساوي لحجم ال Queue، والثاني Outflow rate ويمثل كم Request يمكن معالجته في الثانية الواحدة...

النقاط السيئة لهذه الخوارزمية تتمحور في صعوبة ضبط الأرقام الخاصة بال Bucket Size وال Outflow rate، وسيتم عمل drop لل requests في حال تأخر Request معين ولم تتم معالجته في الوقت المحدد، لأن Rate سيكون حينها limited إن كانت ال Queue ممتلئة...

- Fixed window counter: في هذه الخوارزمية يتم تقسيم ال timeline إلى fix-window (مجموعات بناء على الزمن) ومن ثم يتم عمل Assign لل Counter الخاص بكل window، مع كل Request قادم يقوم بزيادة ال Counter بمقدار 1، إذا لم يكن هناك مساحة كافية لذلك فإن ال Request يتم عمل drop له...شاهد المثال أدناه:

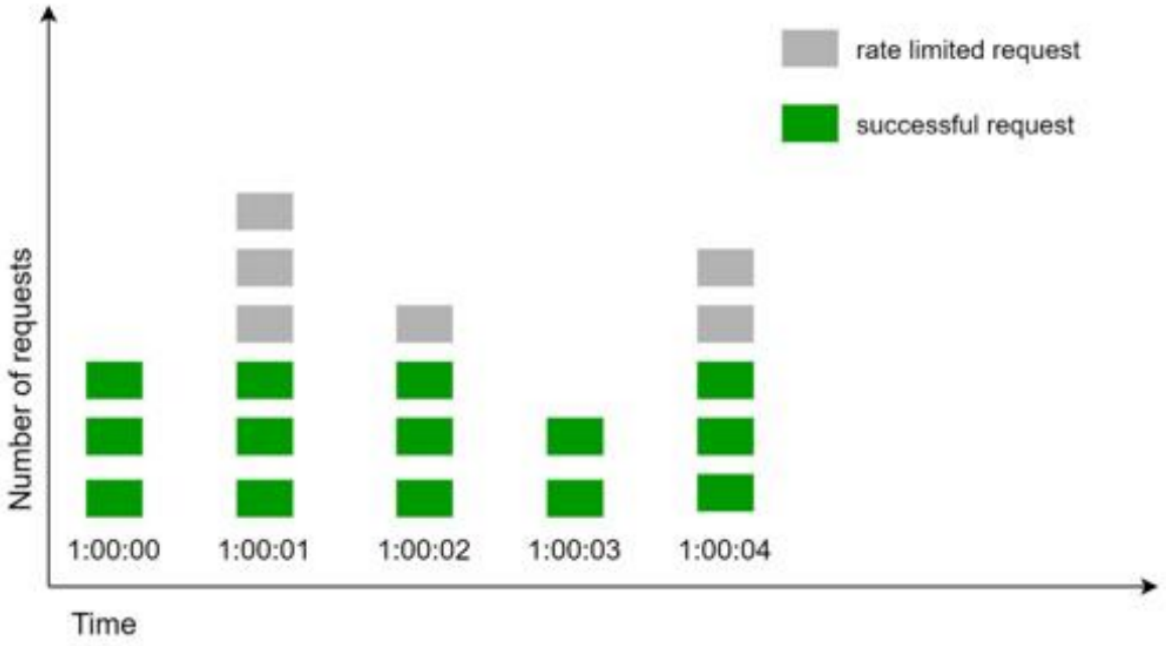
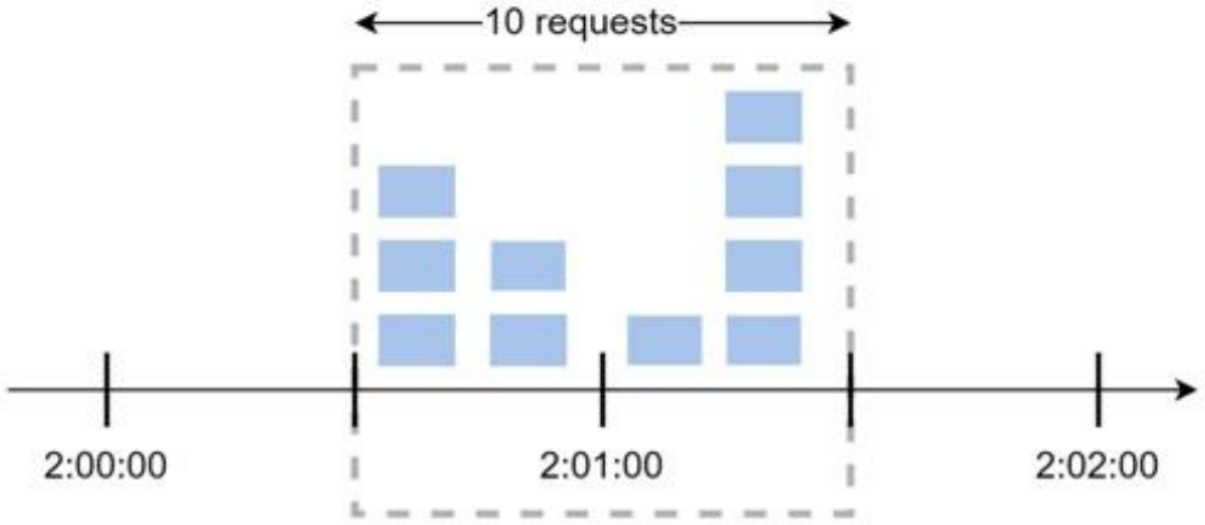


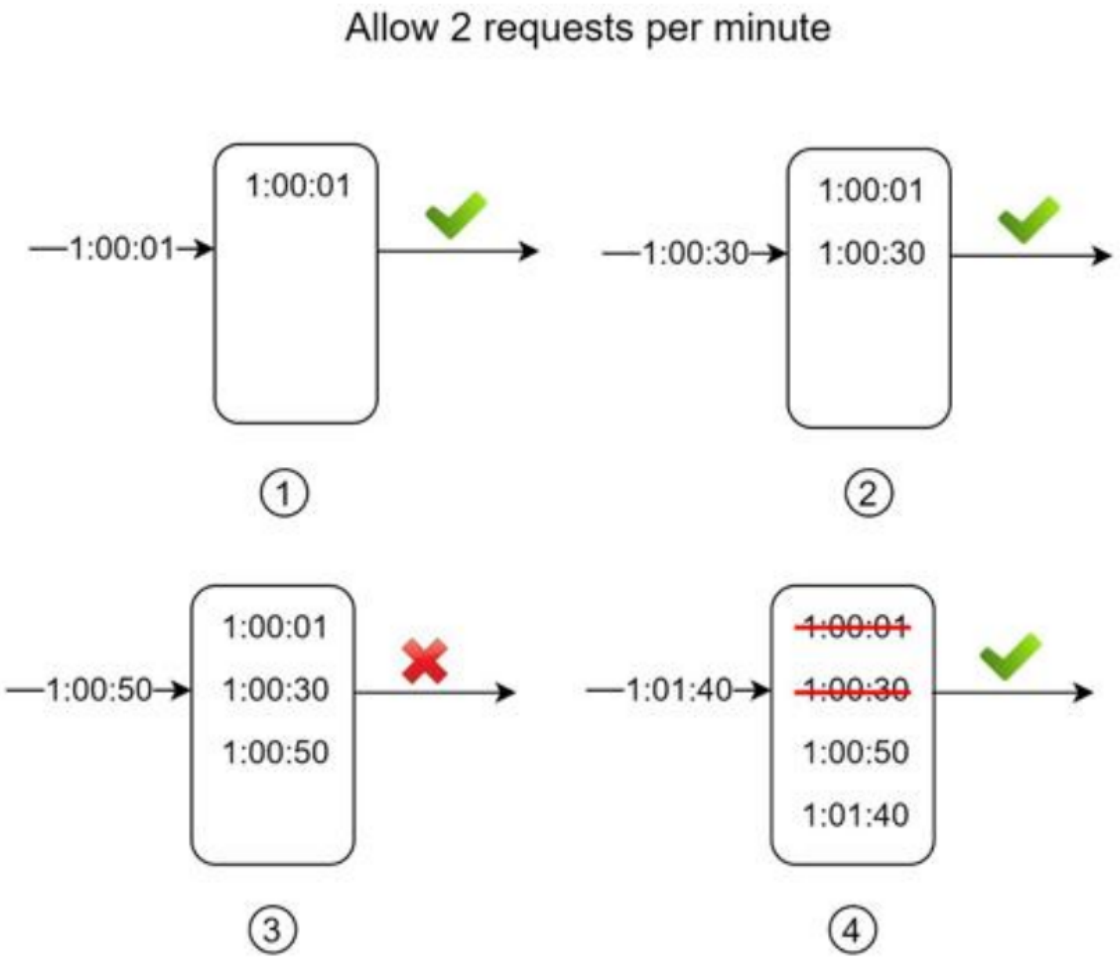
Figure 4-8

في هذه الصورة نلاحظ أن ال timeline تم تقسيمه ليحتمل 3 requests في كل ثانية، أي Request بعد ذلك يعتبر مرفوض لتجاوزه الحد...

مع سهولة هذه الخوارزمية وفعاليتها إلا أن لها سيئة كبيرة، وهي امكانية تحميل حمل أكبر من ال Quota Limit، فمثلا لو سمحت بوجود 5 request في الدقيقة، وجاء 5 request في تمام الساعة 00:01:30 و5 request أخرى في تمام الساعة 00:02:30 فسيكون الناتج 10 request في دقيقة واحدة!، وذلك لأن ال 5 الأولى حصلت في النصف الثاني من الدقيقة الأولى، وال 5 الثانية حصلت في النصف الأول من الدقيقة الثانية...شاهد الصورة:



- Sliding window log: هذه الخوارزمية جاءت لتحل المشكلة الموجودة في الخوارزمية السابقة، ومبدأ عمل هذه الخوارزمية يقوم على حفظ الوقت (timestamp) الخاص بكل request داخل ال Cache (غالبا)، ومع كل request جديد يتم التخلص من ال timestamp log القديمة وإضافة محلها timestamp log جديدة، وفي حال تجاوز الحد يتم عمل reject لل request...



هذه الخوارزمية تتميز بدقتها العالية، فهي لا يمكن أن تتجاوز الحد، لكن لديها سيئة مهمة وهي حاجتها ل Memory كبيرة، والسبب في ذلك يعود لأن كل timestamp rejected تقد تم إضافته إلى ال log أيضا وسيظل موجودا في ال memory...

- Sliding window counter: تعتبر هذه الخوارزمية هجينة بين ال Sliding window log وال fixed window counter، ويمكن تطبيقها بأسلوبين مختلفين، الأول من خلال تطبيق مفهوم يشابه أو يحاكي دحرجة الكرة، فيتم دحرجة الإطار والذي يمثل المدة الزمنية المطلوبة بشكل متتابع لنحافظ على أقصى حد مسموح به من

ال request ضمن هذا الإطار الزمني، ويمكن حساب ذلك من خلال هذه المعادلة: عدد ال Request داخل ال Window الحالية + عدد ال Request في ال Window السابقة * نسبة التداخل بين ال Window الحالية وال Window السابقة والأسلوب الثاني من خلال تقسيم ال Window إلى Buckets كل واحدة منها لهسا سعتها التي يمكنها تحملها بناء على ال Rate limit، مثلا لو قمنا بتحديد حجم ال Bucket ب 20 دقيقة، والحد الأعلى لل Requests مرتبط بالساعة، فسيكون لدينا 3 Buckets، وبذلك يكون لدينا Space Complexity يساوي عدد ال buckets...

High-level architecture

بعد حديثنا العام عن ال Rate Limiter وأشهر الخوارزميات المستخدمة سننطلق الآن للحديث عن المعمارية الخاصة بها، بطريقة أخرى، سنتحدث عن آلية العمل أو التفكير لبناء ال Rate Limiter...

أول ما يجول في خاطرنا هو السؤال التالي: ما هو مكان التخزين لل Counters؟، وبالطبع استخدام قواعد البيانات ليس خيارا جيدا، لأن قواعد البيانات ستكون مخزنة على Disk والتي بدورها ستكون أبطئ!، والحل عادة يكون من خلال استخدام ال In-Memory Cache، وذلك لسببين، السرعة العالية والقدرة على دعم الاستراتيجيات الخاصة بانتهاء الوقت الخاص بال Counters...، ومن الأمثلة المشهورة والمستخدمة بكثرة مع ال Rate Limiters هو ال Redis!، وال Redis هو In-memory Cache يقدم لك Two Commands وهما: INCR و EXPIRE.

يقصد بال INCR ال Command المسؤول عن زيادة ال Counter بمقدار 1، ويقصد بال EXPIRE ال Command المسؤول عن إضافة وقت انتهاء ال Counter، فإذا انتهى الوقت فسيتم حذف ال Counter بشكل تلقائي...، وما سيحدث هنا هو أن ال Client سيقوم بإرسال Request إلى ال Rate Limiter Middleware، وبدوره سيقوم بالتحقق هل ما زال هناك مساحة ل Request جديدة؟، إذا كانت الإجابة نعم فسيتم تحويل ال Request إلى Api Server، وسيتم زيادة ال Counter على Redis، وإذا لم تكن هناك مساحة لذلك، فسيتم عمل Reject لل Request...

لكن، هل هذا كافي؟!، بكل تأكيد لا، فنحن نتحدث عن نظرة مجملة عن الموضوع، ونحتاج للدخول إلى بعض التفاصيل أكثر للإجابة عن سؤالين مهمين، وهما:

1. كيف يتم إنشاء القواعد الخاصة بال Rate Limiter، وأي سيتم تخزينها؟
2. كيف يتم التعامل مع ال Requests التي جاءت وتم استنفاذ ال Rate limit المصرح لها باستخدامه؟

إجابة السؤال الأول تبدأ من ماهية ال Service التي قمت ببنائها، فلو كنت تستخدم مثلا خدمات Google أو AWS فستكون لديك مجموعة من الخيارات المتاحة للتحكم في هذا الأمر، ولو كان لديك ال Server الخاص بك فيمكنك ذلك من خلال العديد من الطرق، واحدة منها استخدام Third party مثل ال Component التي بناؤها بواسطة Lyft، وبكل بساطة كل ما يلزمك هو اتباع القواعد لتحصل على Rate Limiter فعال، شاهد هذا المثال:

```yaml
domain: messaging
descriptors:
  # Only allow 5 marketing messages a day
  - key: message_type
    value: marketing
    descriptors:
      - key: to_number
        rate_limit:
          unit: day
          requests_per_unit: 5

  # Only allow 100 messages a day to any unique phone number
  - key: to_number
    rate_limit:
      unit: day
      requests_per_unit: 100
```

في هذا المثال تم تحديد ال Container الذي يحتوي مجموعة القواعد الخاصة ب Rate Limiter، هذا ال Container هو ال Domain في الصورة أعلاه، ويجب أن يكون هذا الإسم فريد لا يتكرر!، بينما يمثل ال descriptors مجموعة ال key/value المملوكة لهذا ال Domain والتي تحدد ال Rate Limit الصحيح والمناسب لل Request...

في الصورة أعلاه، قمنا بتعريف Container يسمى ب messaging، ووظيفة هذا ال domain وضع حد أعلى لعدد رسائل ال marketing التي يمكن إرسالها في اليوم، مع تحديد الحد الأعلى اليومي للرسائل، وتكون النتيجة أن الجزء الخاص بال message_type markting يسمح له بإرسال 5 رسائل دعائية في اليوم فقط، وفي الجزء الثاني يسمح بإرسال 100 رسالة يوميا فقط لكل رقم هاتف...

أما إجابة السؤال الثاني فتنطلق من معرفة ال Status Code رقم 429، والذي يشير إلى وجود Too many request، بناء على طبيعة النظام الخاص بك، وبناء على شروط معينة يمكنك وضعها للتحقق من حقيقة ال Request القادمة إليك، بحيث يمكنك تجاهل ال

Request تماما، أو يمكنك عمل log له أو يمكنك وضع هذا ال Request في Queue ومن ثم معالجته لاحقا...(سيكون من المهم في كثير من الأحيان للشركات الكبيرة مراقبة ال Requests ومصدرها للتحقق من وجود أي هجمات تخريبية ومحاولة معالجتها قبل أن تحصل)

كما أن ال Client يمكنه أن يعلم عدد المحاولات المسموحة له أو المتبقية له أو متى يمكنه العودة وإرسال طلب وذلك من خلال ال Response Header، فمثلا يقوم ال Rate limiter بإرجاع X-Ratelimit-Remaining ليخبرك بعدد ال Request المتبقية لديك والتي يمكنك إرسالها، ويقوم بإرجاع X-Ratelimit-Limit ليخبرك كم عدد ال Request المسموح لك بإرسالها خلال مدة زمنية محددة، ويقوم بإرجاع

X-Ratelimit-Retry-After ليخبرك كم ثانية تبقى لديك لتتمكن من إعادة إرسال ال Request...

النتيجة لما تحدثنا عنه يمكن تمثيلها بالصورة التالية:

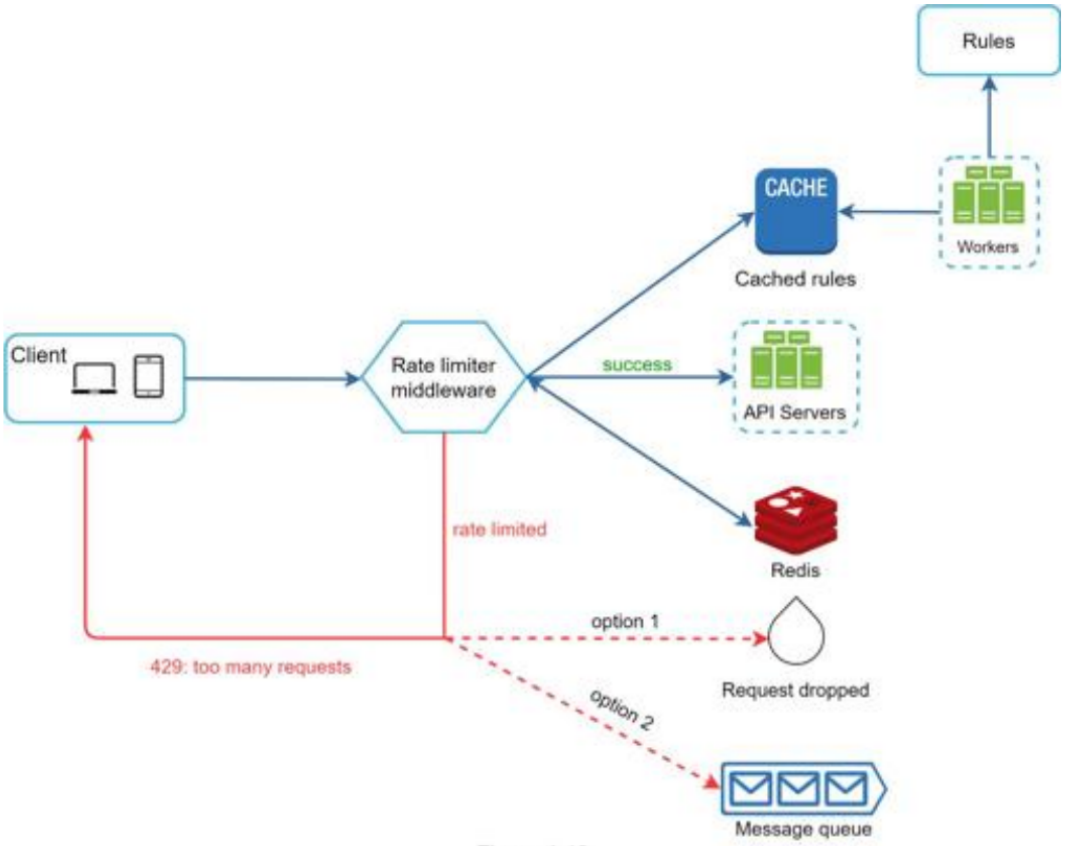


Figure 4-13

شرح الصورة:

يتم أولا حفظ القواعد الخاصة بال Rate Limiter على Disk، ويتم جلبها من خلال workers ومن ثم وضعها بال Cache، يقوم ال Client بطلب Request، هذا ال Request سيمر من خلال ال Rate Limiter Middleware، سيقوم ال Middleware بجلب القواعد من ال Cache وجلب ال Counters وال Expiration time من ال redies، وبعد ذلك سيتم توجيه ال Request إلى ال Api server إذا تبقى للمستخدم مساحة لذلك، أو سيتم رفض ال Request أو وضعه داخل queue لمعالجته لاحقا، وسيتم إرجاع 429 إلى ال Client في هذه الحالة...

Rate limiter in a distributed environment

في حديثنا السابق كان الموضوع سهل وغير معقد، وذلك لأننا نتعامل مع Server واحد فقط، لكن هذا الكلام سيختلف لو كنا نتحدث عن وجود distributed environment، فهناك تحديان يجب أن نكتب الحلول لأجلهما، وهما:

1. Race condition: هذه المشكلة تحدث عند وجود نظام فيه كمية عمليات كبيرة جدا وتحدث بسرعة عالية، فمثلا لديك two request تم إرسالهم من قبل ال Client، وكل thread قام بأخذ Request، ومن ثم تم التحقق من قيمة ال Rate limiter، فإن الإثنان سيتم معالجتهما، مع أن تنفيذ أحدهما قد يجعل ال Rate Limiter يصل إلى الحد، والسبب في ذلك أن التحقق والزيادة صارت بشكل متزامن، لذلك سميت هذه المشكلة بمشكلة السباق، أي تتسابق وتتنافس ال Request ليتم تنفيذها في حين أن أحد هذه ال Request يجب أن يفشل ويفوز الآخر ^^
هناك العديد من الحلول لحل هذه المشكلة، قد يكون أسهلها هو عمل Lock عند قدوم Request، لكن هذا الأسلوب سيجعل من النظام أبطأ وبشكل ملحوظ!، هناك استراتيجيات يمكنك استخدامها لمنع حدوث ذلك ودون عمل lock، مثل استخدام ال sorted set مع ال Redis

2. Synchronization issue: هذه المشكلة أيضا من المشاكل المهمة والتي يجب أن نعيرها اهتماما، فعند وجود الملايين من المستخدمين قد تحتاج إلى بناء أكثر من Rate Limiter Server، في هذه الحالة إذا لم يكن هناك sync بين ال Rate Limiter Server، فسيصبح لكل مستخدم القدرة على تجاوز ال Request * عدد ال Rate Limiter Server في أسوأ الأحوال، وذلك لأن ال Client 1 لو أرسل Request وذهب إلى Rate Limiter 1، ومن ثم قام بإرسال Request جديد وذهب إلى Rate Limiter 2، فسيبدأ أيضا من صفر!، بينما لو تم تحديث البيانات فيما بينهم، فإن هذه المشكلة لن تحدث...(تذكر أننا نتحدث عن stateless)، ولحل هذه المشكلة يمكن بناء Centralized Data store مثل redis وسيتم حل المشكلة ^^

فائدة

الدنيا دار ابتلاء، ومحل الاختبار، وما ينبغي عليك فعله؛ كتابة أفضل ما عندك في هذا الاختبار، لذلك، تصورك الصحيح عن الحياة والغاية منها يجنبك الهشاشة النفسية والضعف النفسي، لأنك تدرك أن ما أصابك من بلاء ما هو إلا مادة من مواد هذا الاختبار، وإن أصابك نعيم فما هو إلا مادة من مواد الاختبار!، فلا تعني إصابة البلاء بأنك تحت العقاب ولا يعني أن النعيم إن أصابك فهو لاستحقاق فيك! وهذا يذكرنا بقوله تعالى: "كُلُّ نَفْسٍ ذَائِقَةُ الْمَوْتِ ۗ وَنَبْلُوكُم بِالشَّرِّ وَالْخَيْرِ فِتْنَةً ۖ وَإِلَيْنَا تُرْجَعُونَ (35)"، فالخير والشر، والغنى والفقر، والعز والذل كلها اختبار لنكتب أحسب أعمالنا وأفضلها...

DESIGN CONSISTENT HASHING

لقد تحدثنا سابقا عن ال Horizontal Scaling، وقلنا أن بناء نظام قابل للتوسع بشكل فعال يعتمد بشكل كبير وأساسي على الأسلوب الذي يضمن توزيع البيانات وال Request بشكل فعال على ال servers المطلوبة، وللقيام بهذه المهمة هناك العديد من الأساليب قد يكون أشهرها أو أكثرها استخداما ال Consistent hashing...لكن، دعونا نتحدث عن المشكلة أولا ثم ننتقل إلى الحل ^^

مشكلة ال Re-hashing: افترض أن لديك أربعة سيرفرات، ولكل server منهم hash معين، ولتحديد أين سيتم تحويل ال Request الخاص بالمستخدم قمنا بتطبيق المعادلة التالية: HashKey % N بحيث تمثل ال N عدد ال Servers، بناء على هذه المعادلة سيتم توزيع ال Request بشكل صحيح ومنظم، والحياة سعيدة طالما أن عدد ال servers لم يزداد أو ينقص!، فمثلا لو اختفى السيرفر رقم 1 لأي سبب، فإن ال Requests الجديدة سيتم تحويلها إلى server مختلف، وذلك حسب المعادلة السابقة!، فستصبح لدينا المعادلة HashKey % 3، وهذا سيتسبب في مشكلة مهمة أن ال Client الذي أرسل Request إلى ال Server رقم 1 سيتم تحويله إلى server مختلف...، وبكل تأكيد هذا يعني أن البيانات التي يطلبها لن تكون متوفرة على هذا السيرفر -مثل فكرة ال Cache server- إلا بتوفير حلول أو طرق لضمان عدم حصول هذا الأمر، وتأكد أنك ستواجه سيل جارف من الأخطاء ^^، ومن هذه النقطة يظهر دور ال Consistent Hashing...

ال Consistent Hashing هو نوع خاص من أنواع ال Hashing، بحيث لو تم استخدامه من قبل ال Hash table وحدث re-size فإن عدد ال keys/slots الموجودة ضمن النطاق أو المتأثرة بالنطاق التي تقع فيه فقط من سيتأثير بالتغيير، لذلك يتم تصنيفه ك distributed hashing scheme وظيفته العمل بشكل مستقل وبغض النظر عن عدد ال Servers أو ال Objects، ويتم ذلك من خلال تحديد مكان لهذا ال Hash ضمن دائرة يطلق عليها hash ring، وبهذا فإن عملية زيادة أو نقصان عدد ال Server لن يؤثر على النظام ككل...باختصار، سيتم حجز مكان لحفظ البيانات ضمن الدائرة فلو تم حذف ال server أو إضافة server فسيكون موقع البيانات ضمن الدائرة هو نفسه وإنما تغير بمن سيرتبط...

مثال إضافة Server:

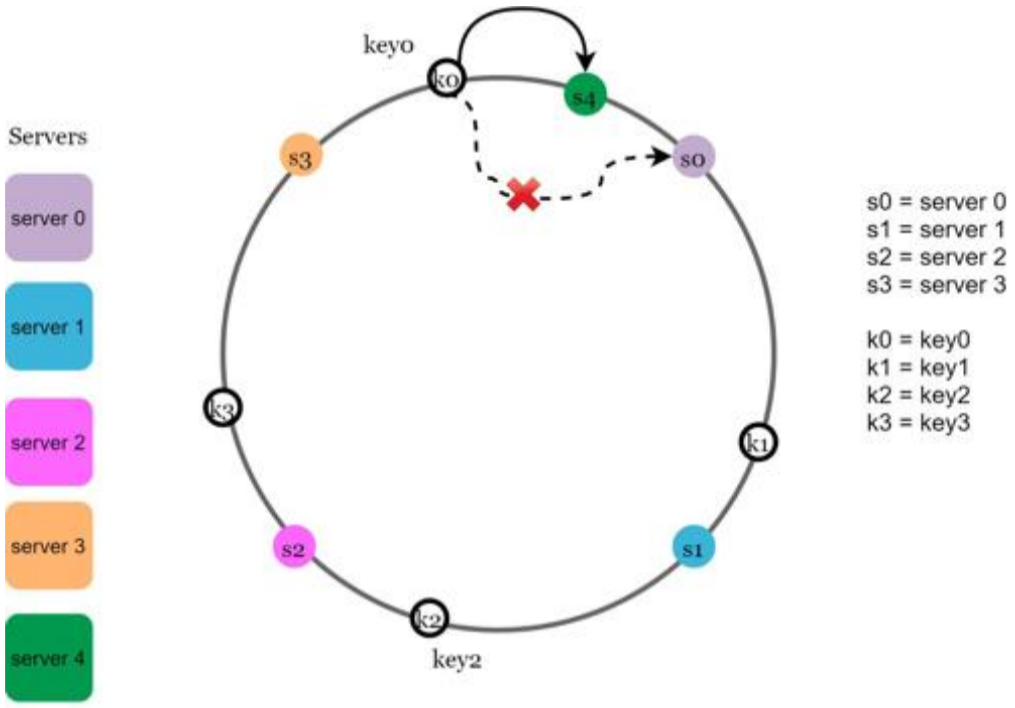


Figure 5-8

لاحظ هنا أن هذه الدائرة تحتوي في داخلها ال Servers وال Keys، إضافة ال Server رقم 4 ألغى الاتصال بين Key0 وال Server 0، وصار الاتصال بين Key0 و Server 4 (يكون الاتصال بين ال Key وأقرب server له)...

مثال 2: عملية حذف السيرفر:

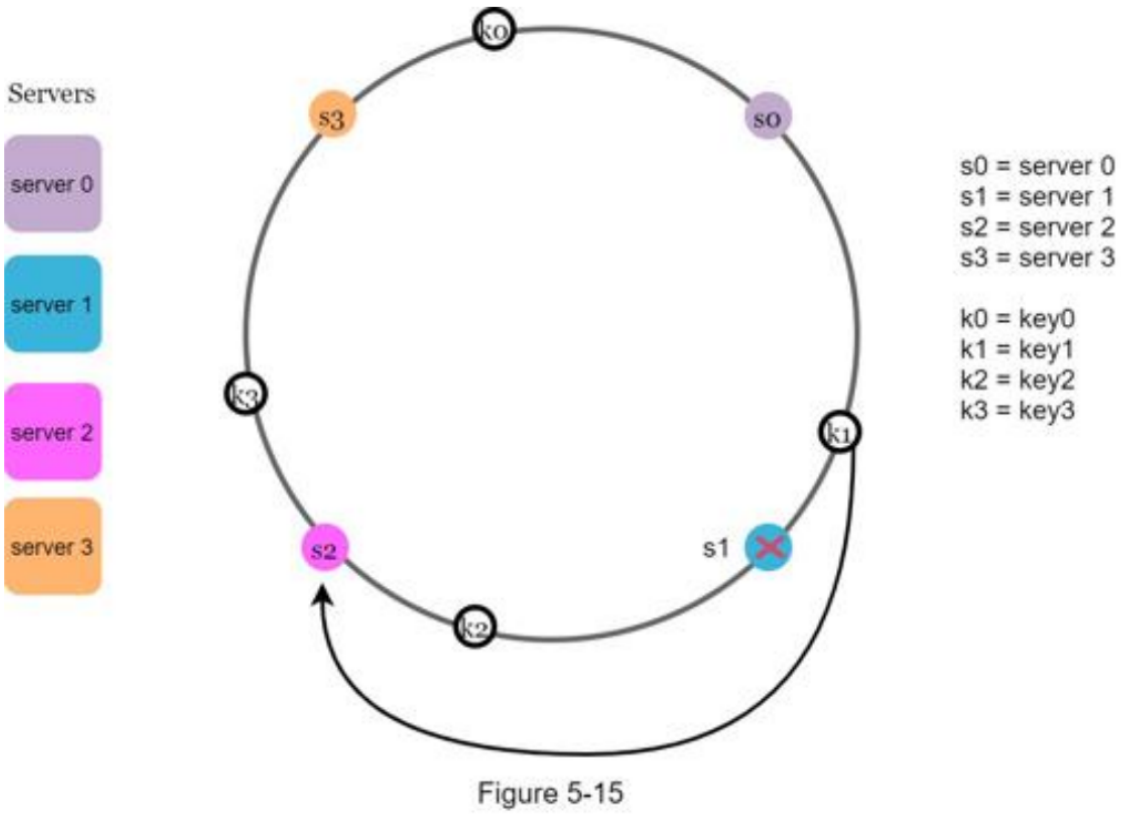


Figure 5-15

لاحظ هنا أن حذف السيرفر رقم 1 أدى إلى جعل الاتصال يتم بين ال Key1 وال Server 2...

فائدة

تأمل في تجارب الأنبياء وما انطوت عليه من الخبرات الدعوية ستجدها تكاد أن تكون جميعا تمثالا ناطقا للصراع بين داعي الوحي الإلهي وفتنة القوة المادية، وستجد افتتان الناس بالقوة المادية يخلب ألبابهم؛ ويعشي أبصارهم، ويصرفهم عن الانصياع والاستسلام للوحي، وستجد العاملين للدين يعانون الأمرين من افتتان الناس بالمظاهر المادية

DESIGN A KEY-VALUE STORE

ال A key-value store هي واحدة من ال non-relational database، يتم استخدام ال key للوصول إلى القيمة، هذا ال Key إما أن يكون HASHED Key وإما أن يكون plan text، بينما يمكن أن تكون القيمة Object, Array, List, String, Number...إلى آخره، ويمكن ان تكون Hashed أيضا!، ومن الأمثلة على هذا Redis وDynamo...، وهناك العديد من العمليات التي يمكن تصميمها وأخذها بعين الاعتبار عند التعامل مع System ستقوم بتصميم Key-Value Store له، وسنتحدث هنا عن ال put(key, value) وال delete(key)...

هذا الشكل key-value يذكرنا بموضوع قديم نستخدمه منذ أن تعلمناه وحتى هذه اللحظة، وهو ال Hash table، وفعليا هذا ما يحدث، فأول حالة لدينا هي التعامل مع ال key-value من خلال Single Server، بحيث تكون العمليات من خلال ال Memory، فسرعتها عالية...، لكن هناك محددات مهمة تتلخص في أن سعة التحمل لل Memory ستنتهي عاجلا أم آجلا مع ازدياد ال Traffic، لكن يمكن تأخير ذلك من خلال ضغط البيانات التي يتم التعامل معها أو من خلال حفظ البيانات المهمة والنشطة في ال Memory فقط، وحفظ الباقي على Disk...، لكن، هذا لن يكون كافيا، لذلك سيتم التوجه لل Distributed key-value store...

Distributed key-value store

قلنا أننا نحتاج لوجود طريقة تمكننا من التمدد ليستطيع النظام تحمل ال Traffic العالي، لذلك كان توزيع ال key-value على أكثر server أمر لا بد منه، وكأننا نتحدث عن Hash table تم توزيعه على أكثر من مكان، لذلك يطلق عليه أيضا: "distributed hash table"...، لكن هناك مصطلح مهم يجب أن ندركه وأن نفهمه جيدا عند حديثنا عن ال distributed system، وهو ال CAP Theorem...

ال CAP هي اختصار ل (**C**onsistency, **A**vailability, **P**artition Tolerance) وهي نظرية تقوم على أساس أن إثنان من أصل ثلاثة من ال CAP يمكن تطبيقها أو الوصول إليها كحد أقصى في الأنظمة الموزعة!، وهذا يعني أنك سقوم بالتضحية بإحدى هذه العناصر في مقابل الحصول على مزيتين أخريين في مقابلها...

وحتى ندرك أكثر ما الذي نتحدث عنه، دعونا نبدأ بتفكيك هذه النظرية:

- **C**onsistency: وهذه تعني إمكانية أن يصل جميع ال Client إلى نفس البيانات في نفس الوقت وبغض النظر عن ال Node التي يتصلون بها، وبمعنى آخر، أن ال Client يجب أن يحصل على أحدث البيانات إذا قام بطلبها، فإذا كانت هناك عملية ما قيد المعالجة، فإن هذا ال Request سيدخل حيز الانتظار لحين انتهاء المعالجة للحصول على أحدث نسخة من البيانات...
- Availability: وهذه تعني أن أي Client قام بإرسال Request data فيجب أن يحصل على Response حتى ولو كانت بعض ال nodes معطلة!، بمعنى آخر القدرة على الوصول وتنفيذ العمليات طوال الوقت...

- Partition Tolerance: ويقصد بها أن النظام ال distributed يجب عليه متابعة ومواصلة العمل بالرغم من وجود أعطال في الاتصال بين two nodes وبغض النظر عن عدد حالات الفشل، بمعنى آخر أن النظام لن يتوقف عن العمل إلا إذا كانت جميع ال network fail...

الآن لنشاهد معا صورة تشرح التقاطع الذي يمكن أن يحصل والتضحية التي يمكن أن تحدث أثناء تطبيق هذه النظرية:

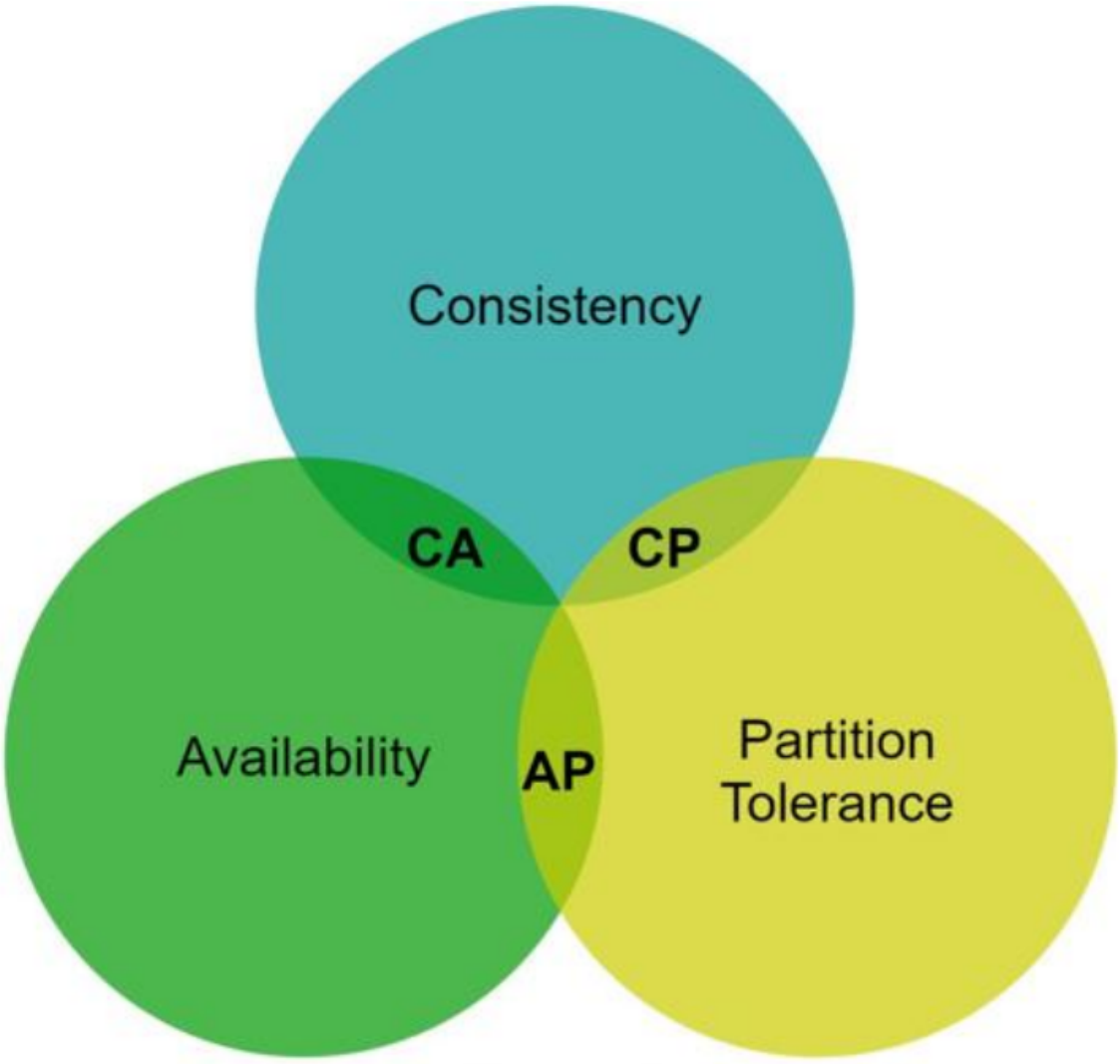


Figure 6-1

في الوقت الحاضر يتم تصنيف ال key-value store بناء على الأحرف الناتجة من التقاطع حسب نظرية ال CAP والتي يمكن أن تدعمها، وهي:

- CA: وتشير هذه إلى النظام الذي يمتلك ال consistency and availability وقام بالتضحية بال partition tolerance، لكن هذا الأمر لا يمكن أن يتواجد في العالم

الحقيقي، لأن إمكانية حدوث فشل في أحد ال Nodes في ال Network واردة جدا...

- CP: وتشير هذه إلى النظام الذي يمتلك ال consistency and partition tolerance وقام بالتضحية بال availability
- AP: وتشير هذه إلى النظام الذي يمتلك ال availability and partition tolerance وقام بالتضحية بال consistency

إذا كنت تعيش في عالم مثالي فحدوث مشكلة على ال Nodes أمر مستحيل ولن يحدث، وبهذا فإن البيانات التي سيتم تخزينها مثلا على Node 1 و Node 2 سنكون متأكدين أنها موجودة ومتزامنة مع Node 3...لكن، في العالم الحقيقي هذه مجرد أحلام، فخروج ال Node 3 من الخدمة سيؤدي لحدوث مشكلة، وحل هذه المشكلة سيكون إما من خلال CP وإما من خلال ال AP، وهذه نقطة مهمة جدا، فطبيعة النظام الخاص بك تتحكم في الأسلوب الذي ستذهب إليه، فعلى سبيل المثال، البنوك أولى أولوياتها أن تضمن تزامن البيانات بأسرع وقت، وأن لا يحدث هناك أي خلل في تزامن البيانات بين أي Node، لذلك فمن الطبيعي أن تختار النظام CP والذي سيتهم بالتزامن وسيضحي بالقدرة على الوصول...، وهذا يعني عمل Block لجميع ال Writes على جميع ال Nodes حتى تعود ال Node الخارجة من الخدمة، بينما إذا قمت باختيار AP، فسيبقى يجلب البيانات حتى وإن كانت قديمة، وسيتم عمل Write على ال nodes التي تعمل هذه اللحظة، وسيتم مزامنة البيانات مع ال Nodes الأخيرة حين تعود للخدمة...

System components

والآن، دعونا نذكر المكونات الأساسية والأساليب التقنية المستخدمة لبناء نظام Key-value store:

- Data partition: في المواقع الكبيرة من غير الممكن وضع جميع البيانات على Single Server، لذلك فتقسيم البيانات على أكثر من سيرفر أمر متوقع، لكن هناك مشكلتين أساسيتين لهذا الموضوع، الأول الحاجة إلى توزيع البيانات إلى جميع ال Servers بشكل معتدل، والثاني الحاجة إلى تقليل كمية البيانات المراد نقلها بين السيرفرات...، وحل هذه المشكلتين يكون من خلال ال Consistent Hasing التي تحدثنا عنها سابقا، وهذا سيعطينا القدرة على إضافة وحذف ال server حسب الحاجة والثاني أن عدد ال virtual node التي يمكن إنشائها ضمن ال Ring Circle سيعتمد على القدرة الخاصة بكل server...
- Data replication: للحفاظ على قدرة وصول عالية ذات موثوقية مرتفعة، يجب علينا عمل Replication للبيانات على أكثر من Server بشكل asynchronously، ويتم تحديد هذا العدد من خلال الإعدادات...
- Consistency: بعد توزيع البيانات في المكون السابق للحصول على قدرة وصول عالية، نحتاج إلى ضمان أن هذه البيانات تم تحديثها على ال Nodes المختلفة، لكن هناك عدة مستويات لموثوقية البيانات يمكننا أخذها بعين الاعتبار بناء على طبيعة ونوع النظام الذي بين يدينا، وسنشير إلى عدد ال replicas بالحرف **N**، ونسشير إلى عدد ال acknowledgment التي تم استلامها لعملية ال Writes حتى يتم اعتبارها عملية

ناجحة بالحرف **W**، وسنشير إلى أقل عدد من ال Response يمكن استلامه من ال replica لتعتبر عملية ال Read ناجحة بالحرف **R**...

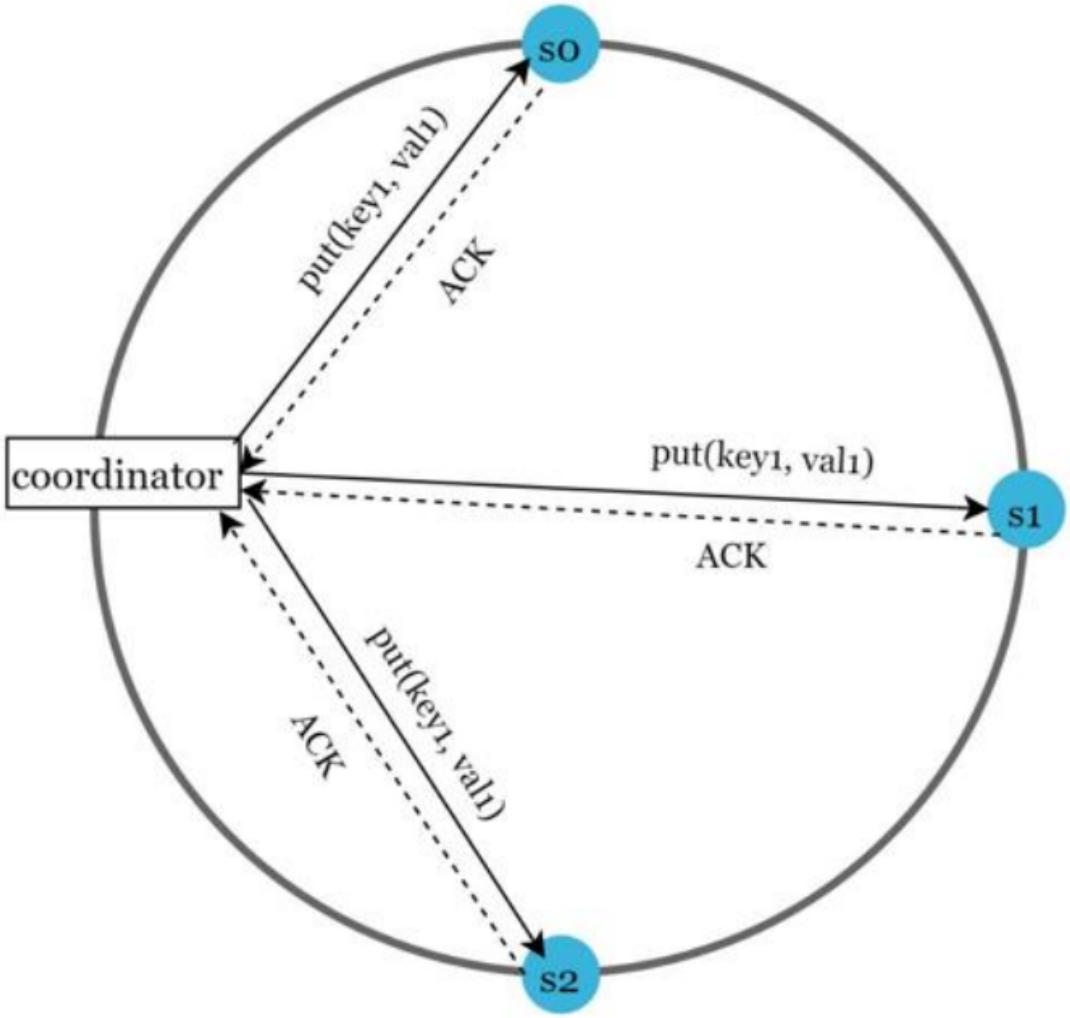


Figure 6-6 (ACK = acknowledgement)

في الصورة أعلاه لو افترضنا أن ال N = 3، وأن ال W = 1، فإن ال Coordinator عليه أن ينتظر ACK واحدة على الأقل حتى تعد العملية ناجحة، فلو قام ال s0 بإرجاع ACK، فإنك لست بحاجة لانتظار ال ACK من ال s1 وال s2...، وهذا يعني أنك هنا أمام خيارين، الأول أن تضحي بالموثوقية مقابل سرعة

الاستجابة أو أن تضحي بسرعة الاستجابة في مقابل الموثوقية!، فلو كان ال W = 1 وال R = 1 فإن السرعة هنا ستكون عالية لأن ال coordinator يلزمه الانتظار ل ACK واحدة فقط من أي replica...ويمكننا تبسيط مجموعة الشروط التي يمكنك استخدامها بناء على طبيعة النظام الخاص بك إلى ما يلي:

- R = 1 & W = N ويكون النظام في هذه الحالة مصمم لغايات القراءة السريعة
- W = 1 & R = N ويكون النظام في هذه الحالة مصمم لغايات الكتابة السريعة
- W + R > N ويكون النظام في هذه الحالة مهتما بالموثوقية العالية للبيانات، وبذلك هناك ضمان بأن البيانات محدثة بالفعل...(لو كان ال N = 3، فإن ال W = 2 وال R = 2 ستحقق الشرط...وهذه موثوقية عالية)
- W + R <= N ويكون النظام في هذه الحالة يقدم ضمانا لموثوقية البيانات، لكن هذا الضمان ليس قويا بما يكفي...

بناء على ما سبق، هناك عدة Models تخص ال Consistency، وهي:

- Strong consistency: وهنا أي عملية Read يجب أن تجلب آخر نسخة محدثة من البيانات، لذلك ال Client لن يرى أي بيانات قديمة!
- Weak consistency: عملية ال Read ليس بالضرورة أن ترجع البيانات المحدثة...
- Eventual consistency: وهو نوع مخصص من ال weak، لكنه يعطي الوقت الكافي ليتم ضمان نشر جميع التحديثات على جميع ال Replica

- Inconsistency resolution: مع أن عملية ال Replication تعطي Availability عالية، إلا أنها أيضا تتسبب في مشاكل على صعيد ال Consistency، لذلك، تظهر لدينا هنا مشكلة ال Inconsistency والتي هي نتاج عمليات التحديث التي لم تتم مزامنتها على جميع ال Replica، ولحل هذه المشكلة يمكنك استخدام ال Versioning & Vector clocks، لاحظ الصورة أدناه لتخيل المشكلة:

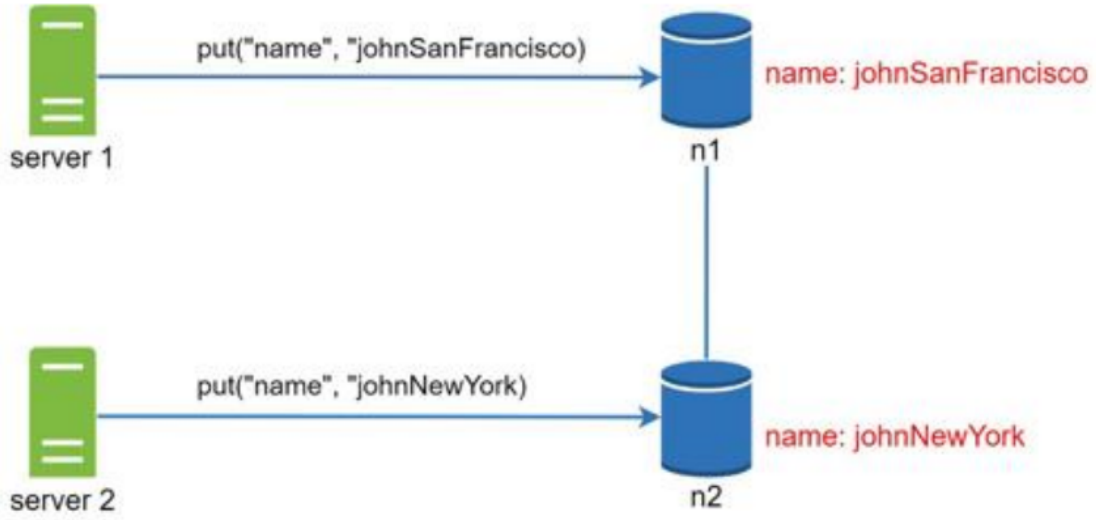


Figure 6-8

في الصورة قام ال Server 1 بعمل تحديث للإسم وتم تخزين هذا التحديث على ال n1، في الوقت نفسه حدثت عملية تغيير أخرى على الإسم من خلال ال server 2 وتخزينها داخل ال n2، في هذه الحالة لدينا conflict بين البيانات الموجودة على n1 و n2...، وحل هذه المشكلة من خلال وضع version لعمليات التحديث، وبعد ال vector clock أحد الأساليب الشائعة لحل هذه المشكلة، ومبدأ عمله قائم على بناء أزواج من البيانات مثل الشكل التالي: [server, version]، بحيث يتم وضع ال

Server المسؤول عن عملية التحديث وال version + 1 لكل عملية تحديث على هذه البيانات، مثال:

D([s0, 1], [s1, 1]), D([s0, 1], [s1, 2])

- Handling failures: حدوث أخطاء في بيئة العمل ال Distributed أمر شائع الحدوث ومتوقع، لذلك تعد عملية معالجة هذه الأخطاء واكتشافها من العمليات المهمة والأساسية لأي نظام، بل وتزداد أهميته بالأنظمة الموزعة...، وهنا لدينا جزءان، الأول هو كيف يمكن أن نكتشف حدوث فشل ما، والثاني كيف يمكننا معالجة هذه الأخطاء (الحلول الشائعة لذلك)...
  - اكتشاف الخطأ: إذا أصبح أحد ال Servers في حالة Down، فهذا الأمر يلزمه تحقق، هذا التحقق لا يكفي أن يكون من خلال Server آخر يخبرنا بأن ال Server في حالة Down الآن...، نحتاج للتحقق من ذلك من خلال أكثر من Server أو بآلية أخرى، لذلك أحد الحلول هو جعل جميع ال Server تتصل مع بعضها البعض وتفصح الاتصال ويطلق عليه all-to-all multicasting، ومع ذلك، فهذا الأمر غير كاف في حال وجود الكثير من ال Servers...، لذلك، هناك حلول أخرى، قد يكون أفضلها هو استخدام decentralized detection مثل ال Gossip بروتوكول، هذا البروتوكول مبدأ عمله بسيط وهو كالآتي:
    - كل Node لديها قائمة بال Node التي ستقوم بالتحقق منها في حال أصبحت Down أم لا، وذلك من خلال وجود ال member ID

وال Heartbeat counter (عداد نبضات القلب الخاصة بال Server) ^^

- كل Node تقوم بزيادة العداد الخاص بنبضات القلب بشكل دوري
- كل Node تقوم بمشاركة عدد نبضات القلب بشكل عشوائي لل Nodes الأخرى، وتقوم ال Node الأخرى كذلك بنفس العملية، ويتم هذا الأمر بشكل دوري...
- بمجرد حصول ال Node على عدد نبضات القلب، يتم تحديث القائمة لأحدث القيم
- إذا لم يرتفع معدل نبضات القلب لمدة زمنية يتم تعريفها مسبقا، فإن هذا ال Server سيتم اعتباره Down

مثال:

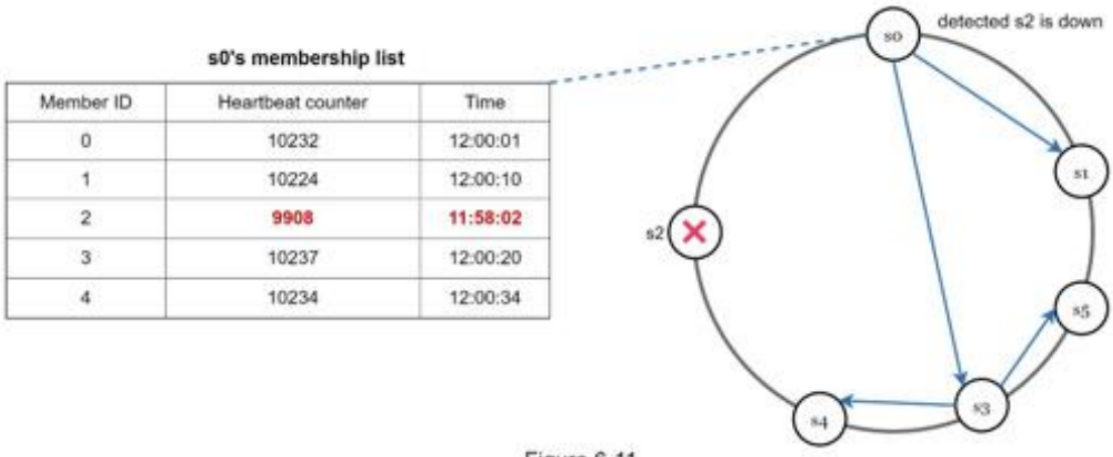


Figure 6-11

في هذا المثال لاحظ أن ال Server 2 في حالة Down، وتم اكتشاف ذلك من خلال ال Gossip، في هذا المثال S0 لديه قائمة بال Nodes وعدد النبضات الخاصة بها،

ولاحظ ال S0 وجود مشكلة عند ال s2 لأن عدد نبضات القلب لم يزداد منذ فترة طويلة، فسيقوم هنا ال s0 بإرسال معلومات ال s2 إلى مجموعة عشوائية من ال nodes، وبمجرد وصول تأكيد من ال nodes الأخرى فسيتم اعتبار s2 في حالة Down...

- معالجة الأخطاء الشائعة أو التعامل معها:

- Handling temporary failures: بعد اكتشاف الخطأ كما تحدثنا في الجزء السابق من خلال ال gossip protocol، نحتاج إلى تطبيق مجموعة من المعايير لضمان ال availability، هذا الأسلوب يطلق عليه ال "sloppy quorum"، هذا الأسلوب فكرته هو تحسين ال availability، فبدلا من تطبيق قوانين ال CAP بشكل صارم، يتم من خلال هذا الأسلوب فك القيود قليلا لزيادة ال availability، ومبدأ عمل هذا الأسلوب بسيط، فبمجرد حصول فشل في ال server يتم النظر إلى أقرب W وأقرب R ما زالت تعمل في hash ring ومن ثم تحويل العمليات إليها مؤقتا حتى يعود ال server للعمل، إذا عاد للعمل، تقوم هذه ال servers بإرسال البيانات إليه لتحقيق ال consistency...(لذلك احذر إذا كان النظام عندك W+R > N)، ويطلق على هذه العملية handoff...
- Handling permanent failures: لقد تحدثنا في الجزئية السابقة عن معالجة الأخطاء المؤقتة، لكن، ماذا لو استمر هذا العطل لمدة طويلة؟، في هذه الحالة يلزمنا أن نفكر في حل يضمن توافقية البيانات بين جميع ال Replica، وهذا الحل يمكن أن يتم من خلال ال anti-entropy

protocol، وفكرة هذا البروتوكول هي مقارنة جميع البيانات الموجودة على ال Replicas وتحديث القديم منها لأحدث نسخة، لكن يجب أن يتم هذا الأمر بطريقة تقلل من كمية البيانات التي سيتم مشاركتها بين ال replicas قد المستطاع، ومن هنا يظهر دور ال Merkle tree (Hash Tree)...، وفكرة هذه ال Tree ببساطة هي عمل Hash لكل leaf وال parent الخاص بها وصولا إلى root، وبهذا سيصبح لدينا hash واحد فقط، وبهذا يمكننا مقارنة هذا ال Hash مع ال Hash الموجود في Replica أخرى، إذا كان كلاهما لديه نفس القيمة، فهذا يعني أن البيانات محدثة، بينما لو كان أحدهما يختلف عن الآخر، سنقوم بعمل مقارنة لل hash الخاص بالجزء الأيسر من ال Replica الأولى مع الجزء الأيسر الخاص بال Replica الثانية...وهكذا حتى نصل إلى محل الاختلاف ومن ثم تحديثه...

- Handling data center outage: احتمالية أن يصبح ال data center خارج الخدمة احتمالية واردة وبقوة، وذلك يمكن أن يحصل للعديد من الأسباب مثل مشاكل في الطاقة أو مشاكل في الشبكة أو مشاكل طبيعية...إلى آخره، لذلك يجب أخذ هذا العطل بعين الاعتبار، ولحل هذه المشكلة، يلزمنا توزيع ال Replica على أكثر من data center...

- System architecture diagram: بعد أن تحدثنا عن ال Design الخاص بال key/value، يمكننا أن نلقي نظرة شمولية على ال Architecture diagram الآن، شاهد الصورة:

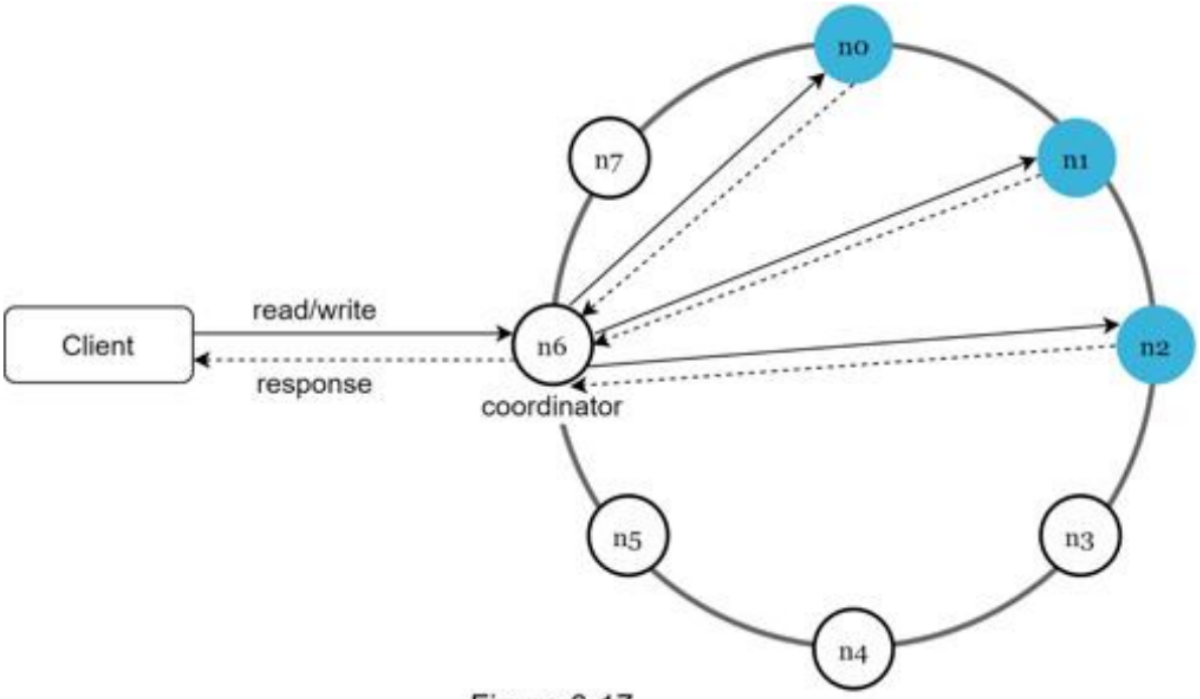


Figure 6-17

بناء على الصورة أعلاه، يمكننا سرد السمات الأساسية لهذه المعمارية:

- يقوم ال Client بالتواصل مع ال key-store من خلال API، من خلال ال get, put...
- coordinator هي node تقوم بالعمل وكأنها Proxy بين ال client وال key-store
- يتم توزيع ال node على ال Ring عن طريق ال consistent hashing
- إضافة وحذف أي Node سيتم بشكل تلقائي ودون أن نقلق...

- سيتم عمل نسخ من البيانات (Data Replica) على أكثر من node...
- لا يوجد هناك Single Failure لأن كل Node لديها نفس المسؤوليات التي لدى الأخرى...
- كل Node من هذه ال Node تقوم بمجموعة من الوظائف وليس وظيفة واحدة مثل التعامل مع ال API ومعالجة الأخطاء وال Replication للبيانات...إلى آخره

- Write path: يتم توجيه الكتابة إلى ال node بشكل مباشر، وبناء على المعمارية المستخدمة لكتابة ال Path تتم الكتابة...
- Read path: يتم توجيه القراءة إلى ال node بشكل مباشر، وبناء على المعمارية المستخدمة لقراءة ال Path يتم جلب المعلومات...

بهذا نكون انتهينا من هذا الفصل والحمد لله ^^، ويمكنكم مراجعة التفاصيل بشكل أعمق أو مراجعتها من خلال هذه الصورة:

Goal/Problems	Technique
Ability to store big data	Use consistent hashing to spread the load across servers
High availability reads	Data replication Multi-data center setup
Highly available writes	Versioning and conflict resolution with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent Hashing
Tunable consistency	Quorum consensus
Handling temporary failures	Sloppy quorum and hinted handoff
Handling permanent failures	Merkle tree
Handling data center outage	Cross-data center replication

Table 6-2

فائدة

الصراع مع الهوى لا يكون مع هوى النفس فقط، بل هناك ما هو أقوى وأخطر على النفس من هوى النفس!، وهو اللحاق بأهواء المخاطبين (ما يطلبه الجمهور)، وهذه مصيبة وطامة تصيب الإنسان...، وهي خفية بحيث لا يدركها إلا من صدق نفسه وفقه منهجه!

DESIGN A UNIQUE ID GENERATOR IN DISTRIBUTED SYSTEMS

أعتقد أننا تعاملنا جميعا مع قواعد البيانات، وأظن أن معظمنا قام بإنشاء جدول ووضع ال Primary key في هذا الجدول هو ال ID، ثم أعطاه خاصية الزيادة التلقائية (auto increment)، لكن، هل سألت نفسك يوما، ماذا لو كان لدي distributed system؟، كيف يمكنني التعامل أو بناء Unique ID على أكثر من قاعدة بيانات؟...هذه الأسئلة وغيرها ستجد الإجابة عنها في هذا الفصل بإذن الله...

للبدء في بناء Unique ID generator يجب علينا أن نسأل أنفسنا مجموعة من الأسئلة المهمة، منها:

1. هل ال ID الذي سنقوم بإنشائه Unique؟
2. هل ال ID الذي سنقوم بإنشائه Sortable؟
3. ما هي الخوارزمية التي سيتم اعتمادها لإنشاء ال ID، هل ال ID + 1؟ أم عن طريق الوقت؟ أم عن طريق عملية حسابية معينة أو أسلوب معين؟
4. هل سيحتوي الـ ID أرقاما فقط؟ أم حروفا فقط؟ أم مزيجا بينهما؟
5. ما هو length المتوقع أو المطلوب لل ID؟ 32bit؟ 64bit؟...إلى آخره
6. ما هو أقل أو أقصى عدد نحتاج لإنشائه في وحدة الزمن؟ مثلا، أقل عدد من ال ID نحتاجه هو 10000 في الثانية؟

بناء على إجابات هذه الأسئلة، يمكنك الانتقال للخطوة التالية، وهي تحديد الخيار المناسب لهذه العملية، ويمكن الاختيار من إحدى هذه الخيارات (مجرد خيارات يمكنك التفكير من خلالها والتوسع لاحقا ^^):

1. Multi-master replication: في هذا الأسلوب نقوم باستخدام ال auto_increment الموجودة في قاعدة البيانات، لكن يتم استبدال مقدار الزيادة من 1 إلى k، وk تمثل عدد ال servers الموجودة، فمثلا لو لدينا Server1 و Server2، فإن ال IDs الموجودة في Server1 ستكون 1 و 3 و 5...إلخ، وال IDs الموجودة في Server2 ستكون 2 و 4 و 6...إلخ، بهذه الطريقة يتم حل **بعض** مشاكل ال scalability، لكن لهذا الأسلوب مشاكل خطيرة وهي:
    a. عملية ال Scale على أكثر من Data center عملية صعبة
    b. IDs do not go up with time across multiple servers.
    c. عند إضافة server أو حذف server فإن عملية ال Scale لن تتم بشكل جيد...
2. UUID (Universally unique identifier): يعد استخدام ال UUID واحدة من أسهل الطرق لإنشاء Unique ID، ال Length الخاص بال ID الناتج هو 128bit -حتى كتابة هذه الكلمات-، وهو ما يعني إنشاء 32 حرفا (لأنه Hex)، مثال: 0621e78e-68ef-4313-8b0e-43b9f3a5c6ed، وأجمل ما فيه هو إمكانية إنشاء ال ID على كل server دون القلق من زيادة عدد السيرفرات أو نقصانها، لأن كل سيرفر يمكنه إنشاء ال ID الذي يحتاجه مباشرة دون الاهتمام بعدد السيرفرات الذي تم إنشاؤه... وبهذا، فإن هذا الأسلوب يتميز بالبساطة والقدرة على

عمل scale بسهولة، والتخلص من أي مشكلة تخص ال synchronization...،
لكن هناك عيوب أيضا وهي:

a. ال Length هو 124bit، وهذا يعني أن هذا الحجم قد يكون كبيرا في بعض الأنظمة أو بناء على متطلباتها
b. ال ID الناتج يحتوي نصوص وأرقام، فلو كانت إجابة الأسئلة المناسبة للنظام الخاص بك أنك بحاجة لأرقام فقط أو نصوص فقط...فلن تستطيع استخدام هذا الأسلوب...
c. لا يرتبط ال ID هنا بوقت إنشائه...، فلن تعرف الوقت الذي يتم إنشاؤه به، مثلا بال auto_increment يمكنك من النظر إلى 1و2و3... أن 3 تم إضافته بعد 2...وهكذا

3. Ticket server: هذه واحدة من الطرق اللطيفة أيضا لإنشاء ال Unique ID، والفكرة هنا ببساطة هي إنشاء Server وظيفته إنشاء ال ID's لل Servers (وهو ما يسمى بال Ticket Server)، وذلك من خلال ال auto_increment الموجودة على قاعدة البيانات الموجودة عليه، ويتميز هذا الأسلوب بأنه ينتج ID ذات طابع رقمي، كما أنه سهل التطبيق ومناسب للمشاريع الصغيرة والمتوسطة...، أما عيوب هذا الأسلوب:

a. غير مناسب للمشاريع الكبيرة، أو يحتاج للكثير من الإعدادات المسبقة
b. Single point of failure، فلو تعطل ال Ticket Server سيتعطل النظام لأن جميع ال Server المرتبطة به لن تعمل، ويمكن حل هذه المشكلة من

خلال إنشاء أكثر من Ticket Server، لكن هذا يحتاج لمجموعة من الإجراءات المهمة للحفاظ على توافقية البيانات...

4. Twitter snowflake approach: هذا الأسلوب تم استخدامه من قبل شركة تويتر، ومشاركته مع المطورين، وفكرته هي بناء Unique ID من خلال بنائه وتجميعه من عدة أجزاء، وهي:

a. Sign bit: وحجمها 1 بت، وقيمتها دوما 0، ويمكن تغييرها في المستقبل إلى 1 إذا حدث هناك طارئ ما أو أمر ما يستدعي فصل الأرقام القديمة عن الجديدة...

b. Timestamp: وحجمها 41 بت، وتمثل الوقت بال Millisecond بالنسبة لزمن معين، مثلا ال Default epoch لتويتر هو 1288834974657 والذي يكافئ = Nov 04, 2010, 01:42:54 UTC، وبناء على حجم هذا الجزء فنحن نتحدث عن ال 2 ^ 41 - 1 وهذا يكافئ 2199023255551، أي أن هذا ال ID سيعمل لمدة 69 سنة في حده الأعلى:

2199023255551ms / 1000s / 365d / 24h/ 3600s

وهذا يعني أن بحاجة للتفكير بحل معين بعد 69 سنة من وجود هذا ال ID ^^ أو بحاجة لإنشاء epoch جداد...

c. Datacenter ID: وحجمه 5 بت، أي يمكن أن يصل ل 2^5 = 32 Datacenter...

d. Machine ID: وحجمه 5 بت، أي يمكن أن يصل ل 2^5 = 32 Machine لكل Data center

e. Sequence number: يمثل رقم متسلسل حجمه 12 بت، يتم الزيادة عليه بشكل متسلسل +1 بالنسبة لل Machine المستخدمة والموجودة في ال Datacenter المستخدمة...، ويتم عمل reset كل جزء من الثانية!، وأقصى رقم يمكن أن يصله هو 2 ^ 12 = 4096، وهذا الرقم يمثل أكبر رقم يمكن كتابته في جزء من الثانية...

شاهد هذه الصورة والتي تمثل ال snowflake:

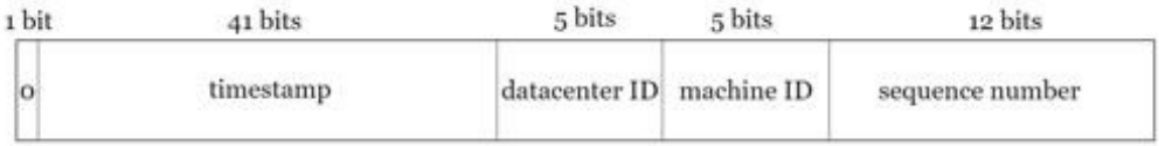


Figure 7-6

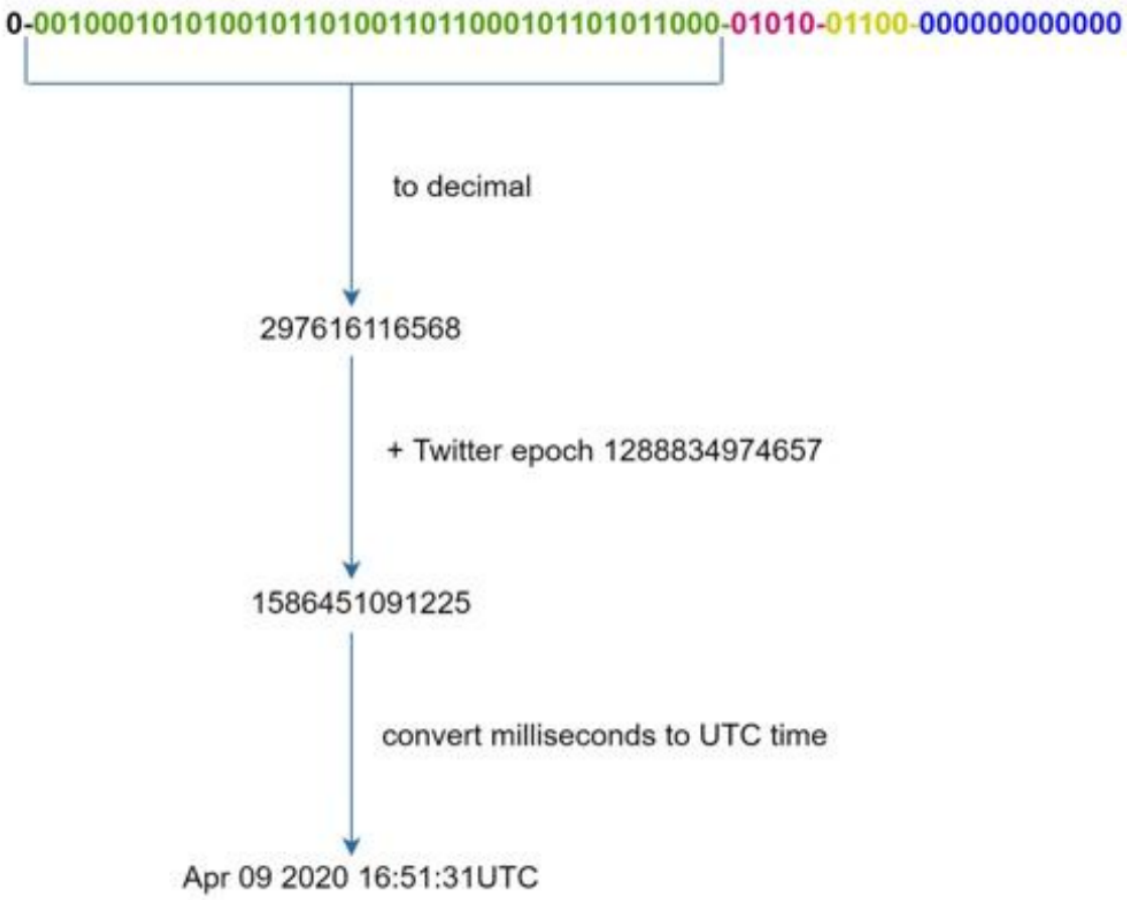

0-00100010101001011010011011000101101011000-01010-01100-000000000000
to decimal
297616116568
+ Twitter epoch 1288834974657
1586451091225
convert milliseconds to UTC time
Apr 09 2020 16:51:31UTC

Figure 7-7

فائدة

إن روح الإسلام إن لامست القلوب والعقول أشعلتها وإن كانت بسيطة!، فالعرب عندما جاءهم الإسلام وهم في قمة البساطة وفي جاهلية، أنار الإسلام أبصارهم وأشعل أفئدتهم فساروا و أناروا العالم كله بحضارتهم، إنهم رأوا أبعد من الطبيعة التي أمامهم، إنهم رأوا عظمة وحكمة أعظم مما في هذه الطبيعة!، إنها الروح التي نقلت البشرية من جهلها وضعفها إلى قوتها، نقلتها من مرحلة إلى مرحلة أخرى تماما!، هؤلاء لم تغرهم فارس وروما، فصاروا أعلاما للبشرية وملهمين لها!

DESIGN A URL SHORTENER

من التجارب الجميلة والتي لابد وأنك استخدمتها من قبل، هي خدمة ال URL shortener، فكم مرة قمت أنت بتحويل رابط طويل وكبير لرابط صغير؟، أو كم مرة قمت فيها بالضغط على رابط ليأخذك إلى رابط مختلف طويل...؟، لكن، هل خطر ببالك يوما أن تقوم بتصميم نظام يقوم بنفس هذه العملية؟، وهل نجحت معك الفكرة؟، وما هي الأفكار التي قمت بتطبيقها والتحديات التي وقعت بها؟...إجابة هذه الأسئلة وأكثر سنحاول الإجابة عنها في هذا الفصل ^^

أول خطوة علينا أن نفهم حدود ونطاق المشروع الذي سنقوم ببنائه، والتي يمكن تحديدها من خلال الإجابة على هذه الأسئلة:

1. هل هناك حد معين لعدد الأحرف المسموح به في الرابط المختصر؟
2. ما هي ال Characters المسموحة في ال short URL؟
3. هل يمكن حذف أو تعديل الرابط المختصر؟
4. ما هو ال Traffic اليومي المتوقع (عدد الروابط التي سيتم تحويلها)؟

بناء على إجابة هذه الأسئلة، يمكننا أن نحلق في هذا النظام، وسنفترض إجابات تناسب احتياجاتنا لنبني عليها النظام الخاص بنا...، وإجاباتها ستكون كما يلي:

1. يفضل أن نصل لأقصر طول ممكن
2. الحروف والأرقام فقط

3. لا يمكن التعديل أو الحذف
4. 100 مليون رابط سيتم إنشاؤه يوميا

مثال عملي:

https://2nees.com/courses-and-articles/design-pattern

النتيجة بعد التحويل:

https://tinyurl.com/3p9vajx3

فاصل معرفي:

الأنظمة الخاصة ببناء الروابط المختصرة تتكون من ثلاثة مكونات أساسية وهي:

1. عملية تحويل الرابط الطويل إلى رابط قصير
2. عملية إعادة التوجيه من الرابط القصير إلى الرابط الطويل
3. الاهتمام بال availability العالية وقابلية التوسع والتسامح مع الأخطاء ضمن اعتبارات معينة...

كما يرتبط بهذه المكونات مجموعة من الأسئلة والإجابات كالتي طرحناها بالأعلى أو التي تحدثنا عنها سابقا مثل ال Scale أو إنشاء ال Unique ID ونحو ذلك، وماهية الإمكانيات أو المتطلبات المرتبطة بالروابط التي يتم إنشاءها مثل الحذف أو التحديث أو expire...ونحو ذلك

الآن بعد أن قمنا بتحديد المتطلبات الخاصة بالنظام، يمكننا البدء بعملية حساب سريعة لما يحتاجه النظام (راجع Back of the envelope estimation)، ويمكن تلخيصها بما يلي:

- لدينا 100 مليون رابط جديد بشكل يومي (Write)، وهذا يعني أننا سنتعامل مع 100 مليون / 24 ساعة / 3600 ثانية = **1160 رابط في كل ثانية**
- سنفترض أن القراءة (Read) اليومية لهذه الروابط هو 10 أضعاف ال Write، فيصبح لدينا 1160 * 10 = **11600 عملية قراءة في كل ثانية**
- عدد ال Records السنوية بناء على هذه المعطيات هو: 100 مليون * 365 يوم = 36.5 مليار Record سنوي...، إذا كان هذا النظام سيعمل لمدة 10 سنوات فإننا سنحتاج القدرة على تخزين ما يقرب من 365 مليار Record
- إذا افترضنا أن الرابط متوسط حجمه هو 100 حرف، فإننا نتحدث عن 100 byte من المساحة التخزينية المطلوبة، فيصبح لدينا 365 مليار * 100 byte يساوي 365TB من ال Storage

من خلال هذه المعطيات العامة، يمكننا الانتقال للجزء التالي، وهو ال High Level Design، وبناء على طبيعة النظام الذي لدينا، فإننا نحتاج إلى ما يلي:

- API ENDPOINT: نحتاج إلى اثنتين من ال APIs، وهما:
  - الأولى لإنشاء الرابط المختصر، وهي من نوع POST يتم من خلالها إرسال longUrl ويتم إرجاع النتيجة من خلالها
  - الثانية GET api وهي المسؤولة فعليا عن قراءة الرابط المختصر وإرجاع الرابط الأصلي لتتم إعادة التوجيه من خلال ال HTTP Redirection

- URL redirecting: إعادة التوجيه في هذا النظام تمثل جزئية مهمة، ومبدأ عملها هو أن السيرفر سيقوم بتحويل الرابط المختصر إلى الرابط الأصلي مع إرجاع ال Status Code رقم 301، وما يحدث بالضبط هو أن المتصفح سيقوم بزيارة الرابط المختصر، ثم سيقوم ال Server بعدها بإرجاع 301 مع كتابة ال Location attribute على ال response header ليصبح مساويا للرابط الأصلي، ثم لأن الخطأ هو 301، فسيتم إعادة التوجيه إلى الرابط الأصلي بناء على القيمة الموجودة داخل ال Location... شاهد الصورة:

Figure 8-1

- URL shortening: وتمثل هذه العملية التي سيتم من خلالها تحويل الرابط الأصلي إلى رابط مختصر، ويلزمنا هنا بناء Hash Function لديه القدرة على تشفير الرابط وفك تشفيره حتى يتسنى لنا القراءة والكتابة بشكل صحيح

من خلال هذه النظرة المجملة، يمكننا الآن الدخول للتفاصيل بشكل أعمق، لأن المعطيات والمتطلبات أصبحت واضحة ^^

إن استخدام ال Hash table هنا مباشرة عملية منطقية من الناحية النظرية -للوهلة الأولى-، لكن سرعان ما ستدرك أن ذلك غير ممكن في الحقيقة، وذلك لأن ال Resource محدودة وغالية الثمن، فلا يمكن الاعتماد على ال Memory لحفظ كمية البيانات الضخمة التي نتحدث عنها، لذلك، عليك اللجوء لقواعد البيانات، وهنا يمكنك استخدام ال Relational Database، فهي ستفي بالغرض، وما نحتاجه هو جدول فيه هذه الأعمدة: id, shortURL, longURL...

الآن، نحتاج لإنشاء الرابط المختصر، وهذا يتم من خلال ال Hash function، وال Hash function هو دالة تستخدم لإنشاء Hash للرابط الأصلي، والنتيجة الخاصة بهذه الدالة تسمى Hash Value...

هذه ال Hash value يجب أن تخضع للمتطلبات التي ذكرناها سابقا، فإذا كان يسمح لهذه القيمة بأن تحتوي على أحرف A-Z,a-z,0-9، فهذا يعني أننا نملك 62 Char ممكن أن يأتي في كل خانة (10 + 26 + 26)، وحسب المتطلبات السابقة، فإننا نحتاج إلى أقل قيمة من طول ال Characters التي يمكنها أن تؤدي الغرض بحيث تكون ال 62 ^ n أكبر أو يساوي 365 مليار، وهذا نجده على 62 ^ 7 = 3.5 تريليون ^^، وهذا الرقم أكثر من كافي، إذا يجب أن ينتج لدينا Hash مكون من 7 Characters...

أما بخصوص ال Hash function، فيمكنك استخدام أي خوارزمية تؤدي الغرض مثل ال md5 أو sha أو base64 وغيرها...، لكن هنا سأقوم بذكر طريقتين من باب التوسع المعرفي، وهما:

- Hash + collision resolution: في هذا الأسلوب يتم عمل Hash من خلال واحدة من الخوارزميات الخاصة بالتشفير مثل ال md5، أو sha1...، بعد عمل ال Hash سنقوم باقتطاع عدد الأحرف الذي نحتاجه ليتوافق مع المتطلبات الخاصة بالنظام وهو 7 أحرف، لكن ستظهر لدينا هنا مشكلة!، فيمكن أن يحدث هناك تصادم بين رابطين مختلفين اشتركوا بنفس البداية لل 7 أحرف، والحل يكون من خلال إضافة predefined string لأجل هذا الغرض...، ولكن هذا الحل سيسبب لنا مشكلة أخرى وهي أننا بحاجة لطخ Query مع كل عملية Hash للتأكد من أن ال Hash value غير موجودة بقاعدة البيانات...
- Base 62 conversion: يعد ال Base conversion واحد من أشهر الأساليب المستخدمة مع ال url shortener، فهو يقدم قاعدة تمكنه من تحويل الرقم نفسه لشكل آخر، وفي مثالنا هنا Base 62 لديه القدرة على التعامل مع 62 حرف، وهو مناسب لطبيعة المتطلبات الخاصة بنا أيضا...^^، وطريقة عمله كما يلي:
  - لدينا خريطة من الأحرف شكلها كالآتي: 0-0, 1-1, وصولا إلى 9-9، بعدها A-10 ثم B-11...حتى نصل إلى z-61، وهذا يعني أن الرقم 10 سيمثل A والرقم 61 سيمثل z...

وهذا جدول يوضح الفكرة:

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9
10	A	11	B	12	C	13	D	14	E	15	F	16	G	17	H	18	I	19	J
20	K	21	L	22	M	23	N	24	O	25	P	26	Q	27	R	28	S	29	T
30	U	31	V	32	W	33	X	34	Y	35	Z	36	a	37	b	38	c	39	d
40	e	41	f	42	g	43	h	44	i	45	j	46	k	47	l	48	m	49	n
50	o	51	p	52	q	53	r	54	s	55	t	56	u	57	v	58	w	59	x
60	y	61	z																

- لو افترضنا أن لدينا ال id رقم 11157، فإن تشفيره سينتج لدينا 2tx، وهذا ناتج باقي القسمة لل id:
  - 11157 % 62 = 59 وناتج القسمة 179
  - 179 % 62 = 55 وناتج القسمة 2
  - 2 % 62 = 2 وناتج القسمة 0

بناء على نتائج الأرقام، فإن ال 2 تبقى كما هي، وال 55 يعادلها t وال 59 يعادلها x، فيصبح لدينا 2tx

بعد هذه التفصيلات، يمكننا وضع مقارنة سريعة بين الأسلوبين اللذين تحدثنا عنهما، وهي:

Hash + collision	Base 62

طول التشفير الناتج غير ثابت، ويزداد أو يتغير بازدياد أو تغير القيمة	طول التشفير الناتج ثابت
عملية التشفير تحتاج إلى عملية إنشاء Unique Id قبل التشفير	لا تحتاج إلى إنشاء Unique Id
لا يمكن أن يحدث هناك تصادم (تكرار) لل Hash بقيم مختلفة	يمكن أن يحدث هناك تصادم (تكرار) للقيم المختلفة، ذلك يلزم هنا الحذر واتخاذ الإجراء اللازم عند حدوث هذه المشكلة
يمكن توقع أو معرفة الرابط المشفر التالي بناء على عملية التشفير، وهذا قد يتسبب في وجود مشكلة أمنية أو على مستوى الخصوصية...مثلا لو افترضنا أن 11157 هي 2tx فيمكنني معرفة الرابط التالي المخزن في قاعدة البيانات من خلال زيادة 1 على الرقم ليصبح 11158	لا يمكن معرفة الرابط المشفر التالي أو السابق من خلال ال Hash، لأن عملية إنتاج ال Hash لا تعتمد على ال ID

ملاحظة: عند استخدامك لأسلوب مثل ال Base 62، فعليك الانتباه لموضوع التوسع واستخدامه في ال Distributed system، وهذا الأمر لقد تحدثنا عنه في الفصل السابق بحمد الله ^^

فائدة

لا يجدر بالكفاح ومكافحة الاحتلال أن تكون فقط نابعة من شعور بطولي!، بل يجب أن تكون نابعة من معنى أسمى وهو لا إله إلا الله!، فمن كان هذا طريقه علم أنه يجب أن يقدم شيئا، شيئا عظيما يختلف عما يقدمه من لا يؤمن بالله!، إنك صاحب رسالة، وصاحب الرسالة عليه إعلانها وإنقاذ نفسه وغيره ما قدر له ذلك واستطاع!

DESIGN A WEB CRAWLER

يعد ال Web Crawler واحد من الأنظمة الجميلة والممتعة التي قد تحتاجها يوما ما، فمن خلالها يمكن الحصول على الكثير والعديد من ال Content الخاص بالصفحات أو التحقق منها أو دراستها وجمع تحاليل من خلالها، وال Content تشمل كل ما يمكن أن يكون بال Web Page مثل الصور والنصوص والفيديوهات ونحو ذلك

يعمل ال Web Crawler من خلال فكرة بسيطة، وهي البدء بجمع مجموعة من الصفحات ومن ثم قرائتها والبحث عن صفحات أخرى من خلال هذه الصفحة، فمثلا لو دخل موقع 2nees.com ووجدت بداخله رابط إلى 2nees.com/css فسيتم جلبها ومن ثم البحث عن رابط آخر بداخلها وهكذا...، وهذه التقنية مفيدة جدا وتقوم عليها واحدة من أهم التطبيقات حاليا، وهي خدمات ال Search Engine مثل Google!، فجوجل يحصل على الصفحات ويقوم بعمل ال indexing لها من خلال هذه التقنية، ويقوم بهذه الوظيفة Googlebot ^^، كما تستخدم هذه التقنية لأخذ نسخ وأرشفتها زمنيا مثل موقع ال archive، ويستخدم أيضا في البحث عن الصفحات التي انتهكت حقوق الملكية، ويستخدم أيضا في عمليات التنقيب عن البيانات (Data mining) والتي تقدم خدمة رائعة من خلال الحصول على بيانات مهمة بناء عل اهتمامات الأشخاص مثلا ونحو ذلك، ومثال هذا الشركات المالية التي تتابع اهتمامات الناس لتوقع ارتفاع سعر أسهم شركة معينة أو انخفاضها...

ويعتبر تطبيق ال Web Crawler عملية سهلة تزداد تعقيدا بازدياد التعقيد والشروط المطلوبة للحصول على المحتوى، لذلك قد تجد تطبيقا يتم الانتهاء منه خلال 4 ساعات وآخر يبقى قيد التطوير بلا توقف ^^

آلية تطبيق مفهوم ال Web Crawler بشكل مبسط:

1. يتم إدخال أو وضع مجموعة من الروابط المراد تحميلها وأخذ البيانات منها
2. البحث عن جميع الروابط الموجودة في هذه الصفحات
3. إضافة الروابط التي تم إيجادها في النقطة رقم 2 ليتم تحميلها...
4. تكرار نفس الخطوات مجددا...

آلية التطبيق البسيطة هذه هي الفكرة العامة التي ينطلق ال Web Crawler منها للعمل، لكنها بالتأكيد ليست بهذه البساطة خصوصا إذا تحدثنا عن أنظمة كبيرة أو معقدة مثل محركات البحث، فالتفاصيل والشروط التي يمكن أن يتم وضعها ضمن هذه الخطوات وترتيبها سيؤثر على مستوى التعقيد الخاص بال Web Crawler، ومع ذلك، تبقى الفكرة الأساسية التي تنطلق منها هي المحور الذي سيقودك لبناء الفكرة المعقدة...

هناك العديد من النقاط التي يجب أن تضعها بعين الحسبان عند قيامك ببناء ال Web Crawler لتخرج بنتيجة جيدة مثل:

- Scalability: عدد الصفحات التي يمكن أن يزحف من خلالها ال Web Crawler عدد كبير جدا، يتجاوز المليارات من الصفحات، لذلك يجب أن يعمل بكفاءة عالية، وهذا يتم من خلال ال parallelization.
- Robustness: عند تعاملك مع العديد من الصفحات والتي بدورها توجد على العديد من الخوادم وال Data centers ونحو ذلك، عليك أن تضمن متانة ال Web Crawler الخاص بك، وهذا يتم من خلال القدرة على التعامل مع حالات الأخطاء الممكنة، مثل أن ال Server down أو أن الصفحة لا تعمل أو أن هناك روابط ضارة ونحو ذلك
- Politeness: يجب أن يكون ال Web Crawler الخاص بك "مؤدب" ^^، ويقصد بهذا الكلام أن تقوم بعدد منطقي من ال Requests في مدة زمنية قصيرة، فكثير ممن يقوم ببناء Web Crawler يقوم بإطلاق الكثير من ال Requests في وقت قصير مما يؤدي إلى تعطل الخدمة أو جعلها أسوأ لصاحب هذه الصفحات، أو سيتم طردك باعتبارك "مخرب" ^^
- Extensibility: يجب أن يكون ال Web Crawler الخاص بك لديه القدرة على التوسع ليشمل خصائص ومميزات أخرى مثل إمكانية إضافة خاصية دعم سحب الصور لو كنت تسحب النصوص فقط، وإمكانية إضافة شروط جديدة تخص الصفحات أو المحتوى الخاص بها ونحو ذلك...

نماذج من الأسئلة التي يجب أن تسألها لنفسك عند البدء بتصميم ال Web Crawler:

- ما الهدف الأساسي لبناء هذا ال Web Crawler؟ ولنفترض أنه Search Engine Indexing
- ما هو عدد الصفحات المتوقع جمعها من خلال هذا ال Web Crawler شهريا؟ ولنفترض أنها 1 مليار صفحة
- ما هي أنواع المحتوى التي نرغب في جمعها؟ ولنفترض أنها صفحات ال Html
- هل علينا الاهتمام بالصفحات التي تم تعديلها أو عند إضافة صفحات جديدة؟ ولنفترض أن الإجابة نعم
- هل هناك حاجة لحفظ الصفحات التي تمت زيارتها وجمع محتواها لدينا؟ لنفترض نعم، ولمدة 5 سنوات
- في حال وجود صفحات مكررة، ما هو الإجراء المطلوب القيام به في هذه الحالة؟ لنفترض أننا سنتجاهل الصفحات المكررة

بناء على الأسئلة السابقة، يمكننا البدء بوضع المتطلبات التي يحتاجها النظام كما تعلمنا سابقا (BACK-OF-THE-ENVELOPE)

افترضنا أن مليار صفحة سيتم تحميلها شهريا من خلال ال Web Crawler، بناء هذا على الافتراض فإن ال

- QPS = 1000000000 / 30 Day / 24 Hour / 3600 = ~400 Page per second

- إذا افترضنا أن الصفحة حجمها 500K فإن مساحة التخزين المطلوبة شهريا هي:
  1000000000 * 500K وهذا يساوي 5TB شهريا
- هذه الصفحات سيتم حفظها لمدة 5 سنوات، إذا 5 * 12 * 5TB وهذا يساوي 30 PB تقريبا ^^

بناء على هذه المعطيات، استطعنا توقع ما نحتاجه لل Web Crawler الخاص بنا...والآن ننتقل إلى الجزء التالي، وهو وضع ال High Level Design، شاهد الصورة:

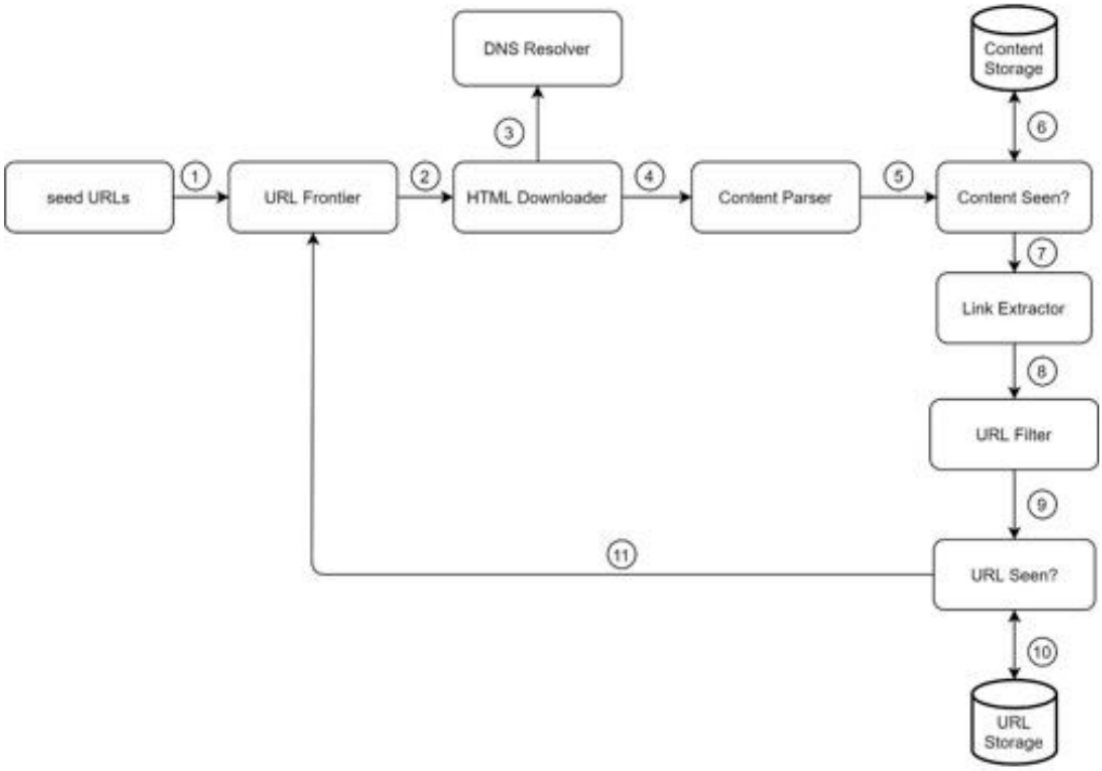


Figure 9-4

في الصورة المرفقة أعلاه، يظهر لدينا المكونات التي نحتاجها، وهي:

- Seed URLs: تمثل هذه المرحلة نقطة البداية لل Web Crawler، وهي النقطة التي يتم وضع ال Starting Point لل Web Crawler لينطلق بعدها بجلب الصفحات الأخرى، فمثلا أول رابط يمكن أن نضعه هو رابط الموقع الخاص بنا مثل: 2nees.com، ومن هنا سنحاول الوصول إلى أكبر قدر ممكن من الصفحات الخاصة بالدورات والمقالات، لكن هنا لدينا تحدي مهم، وهو ما هي مجموعة القواعد التي يمكن أن نضعها لهذه البداية، فكلما قمت بوضع مجموعة مميزة من ال Start Point فهذا يعني الحصول على نتائج أفضل، مثلا، هل هذا ال Domain لديه أكثر من جزء يعمل على جلب محتوى مختلف حسب الدولة؟ وماذا عن ال Category الخاصة بالموقع، هل يمكننا الوصول إلى مجموعة ال Category الموجودة في هذا الموقع والزحف على أكبر قدر ممكن من الروابط داخل هذا التصنيف أم لا؟...إلى آخره، واحدة من أشهر الأفكار لمعالجة هذه المشاكل هي تقسيم (Divide) الرابط إلى أجزاء صغيرة، فمثلا نتأكد من أن الجزء الأول هل يبدأ برمز دولة مثل JO أو KSA...إلى آخره، لا يوجد هنا إجابة محددة للأسلوب الذي ينبغي عليك استخدامه، بل الإجابة هنا هي ما تحتاجه للحصول على أفضل نقطة للبداية!
- URL Frontier: في الأساليب العصرية لبناء ال Web Crawler يتم فصله إلى جزئين، الأول جاهز للتحميل، والثاني تم تحميلها بالفعل!، ال URL Frontier هي الجزء الخاص بحفظ الروابط الجاهزة للتحميل، ويمكن استخدام ال FIFO لهذا الغرض...
- HTML Downloader: هذا الجزء هو المسؤول عن تحميل الصفحة من الإنترنت بعد أن يقدم ال URL Frontier الرابط له

- DNS Resolver: يتم تحويل ال Domain name إلى IP address
- Content Parser: هذه ال Component وظيفتها معالجة المحتوى الذي تم تحميله والتحقق من طبيعة المحتوى وخلوها من المشاكل الأمنية المحتملة، وهذه الجزئية تستغرق بعض الوقت، لذلك يتم وضعها باعتبارها جزئية مستقلة بذاتها ضمن ال flow
- Content Seen: في هذه الجزئية يتم التحقق من المحتوى الذي تم جلبه، هل هو موجود عندنا من قبل؟، إذا كان موجودا فلا داعي لإضافته، فتقريبا 29% من المحتوى الموجود على الإنترنت مكرر!، وهناك العديد من الطرق لمقارنة المحتويات القديمة والجديدة فيما بينها، منها أن تقارن ال Html القديم بالجديد حرف حرف!، وهذه العملية طويلة وغير مجدية فعليا!، ماذا لو قمت بعمل Hash للمحتوى القديم والجديد وقارنت النتائج؟! ^^
- Content Storage: في هذه الجزئية يتم تحديد المكان والآلية المناسبة لحفظ البيانات، وهذا يعتمد على طبيعة البيانات التي ستقوم بجلبها وحجمها، وغالبا ما يتم حفظ المحتوى على ال Disk بسبب كبر حجمه، ويتم الاحتفاظ بنسخة من المحتوى الشائع أو المطلوب بكثرة على ال Memory
- URL Extractor: يتم في هذه المرحلة جلب الروابط الموجودة في الصفحة وتحويل ال Relative Path إلى Absolute Path، مثل تحويل /css إلى https://2nees.com/css
- URL Filter: توجد هنا مجموعة من الشروط التي سقوم بإزالة الروابط غير المرغوب بها مثل الروابط التي لا تعمل أو المسجلة ضمن القائمة السوداء

- URL Seen: هنا يتم التحقق من الرابط هل تم زيارته من قبل أو أنه في URL Frontier أم لا...
- URL Storage: هنا يتم حفظ الروابط التي تمت زيارتها

الآن بعد أن فهمنا المكونات الأساسية بشكل عام، علينا أن ندخل بشكل أعمق في بعض التفاصيل المهمة لبناء الأجزاء السابقة، وهذه التفاصيل يمكن تقسيمها إلى:

- Depth-first search (DFS) vs Breadth-first search (BFS): عند تفكيرك بال Web ستجد أنك تتعامل مع Graph بحيث تمثل ال Web Pages ال nodes وتمثل ال URLs ال edgis، وأنت ستقوم بالعبور من Web Page إلى Web Page من خلال استخدام ال URL، لذلك سيخطر على بالك خوارزميتين يمكنهما تنفيذ هذه المهمة، وهما ال BFS وال DFS، لكن يعد استخدام ال DFS هنا خيار سيء، والسبب في ذلك أن Depth الخاص بال DFS قد يكون Very Deep (يمكنك النظر إلى طريقة عمل الخوارزمية ومقارنتها بال BFS من هذه الناحية، تخيل وجود Tree عندما تصل إلى نهايتها ستضطر للعودة للبحث عن neighbors لم تتم زيارتها ثم قارن ال Longest Path)، لذلك يعد ال BFS الخيار الأكثر شيوعا من خلال استخدامه مع ال FIFO، لكن هناك مشكلتان لهذا الأسلوب، وهما:
  - impolite: الروابط الموجودة في نفس الموقع ستقوم بالإشارة لنفس الموقع، وهنا سندخل في عملية معالجة كبيرة تستغرق ال Recourse على نفس ال host، وبهذا ستكون جميع ال Request التي يتم اطلاقها بشكل parallel

ستخدم نفس ال host بدلا من أن تخدم أكثر من host..مشكلة "Politeness"

- priority: في ال Standard BFS لا يوجد آلية لمراعاة الأولوية الخاصة بالروابط المراد جلبها، فالصفحات لا تتساوى في أهميتها، فقد تجد صفحات بالكاد تتم زيارتها وصفحات مشهورة وشائعة، بمعنى آخر لا يوجد هناك آلية بالشكل الافتراضي للإعتناء page ranks, web traffic, update frequency..إلى آخره

- URL frontier: هذه ال Component هي المسؤولة عن حل المشاكل السابقة، فهي تمثل ال Data Structure التي تقوم بحفظ الروابط ليتم تحميلها، فمشاكل مثل ال Politeness وال URL prioritization وال freshness يمكن معالجتها هنا قبل تحميل الصفحات ^^، ولنشاهد هذه الحلول:
  - Politeness: بشكل عام يجب على ال Web crawler أن يمنع إرسال عدد كبير من ال Requests على نفس ال host في مدة زمنية قليلة، وكما تحدثنا سابقا فهذا أسلوب غير مهذب وقد يتم اعتباره هجوم من نوع DOS، ولحل هذه المشكلة يمكنك ببساطة منع تحميل أكثر من صفحة ضمن وقت معين لنفس ال host، مثلا يمكنك السماح بتحميل صفحة واحدة من موقع 2nees.com كل 5 ثواني، لكن كيف يمكن القيام بهذه العملية؟، يمكنك

ذلك من خلال مشاهدة هذه الصورة:

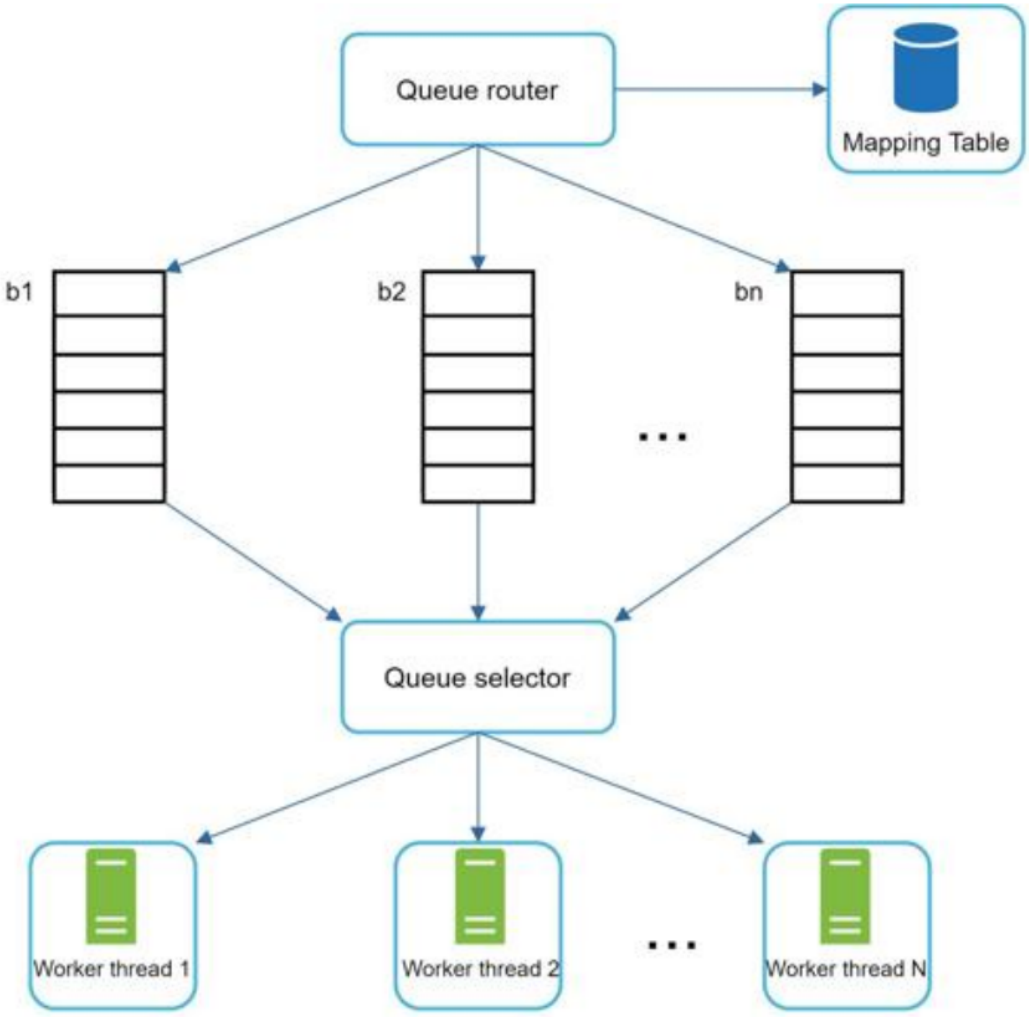


Figure 9-6

- في الصورة أعلاه لدينا Queue Router، وظيفته التأكد من أن كل host موجود داخل Queue محددة فقط، مثلا 2nees.com موجود داخل ال Queue b1 فقط، و example.com موجود داخل b2 فقط وهكذا...، وتكون وظيفة ال Mapping Table هي عمل map بين ال Queue وال host، وكل Queue تعمل بآلية ال FIFO،

ويقوم ال Selector باختيار الرابط المراد تحميله بناء على الشرط / الشروط المطلوبة، ويمكن إضافة الوقت بين كل عملية تحميل على مستوى ال Worker Thread، مثلا Worker Thread 1 سيقوم بتحميل 2nees.com/page1 و 2nees.com/page2 لكن سيكون هناك فارق زمني مقداره 3 ثواني لذلك...

- Priority: تحدثنا سابقا أن الصفحات ليست بذات الأولوية أو الأهمية، وأننا نحتاج إلى آلية ما تساعدنا على عمل Crawler للصفحات بناء على أهميتها وأولوياتها، وفي هذه المرحلة يمكننا الدخول بعمق أكثر إلى هذه ال

Component، شاهد الصورة:

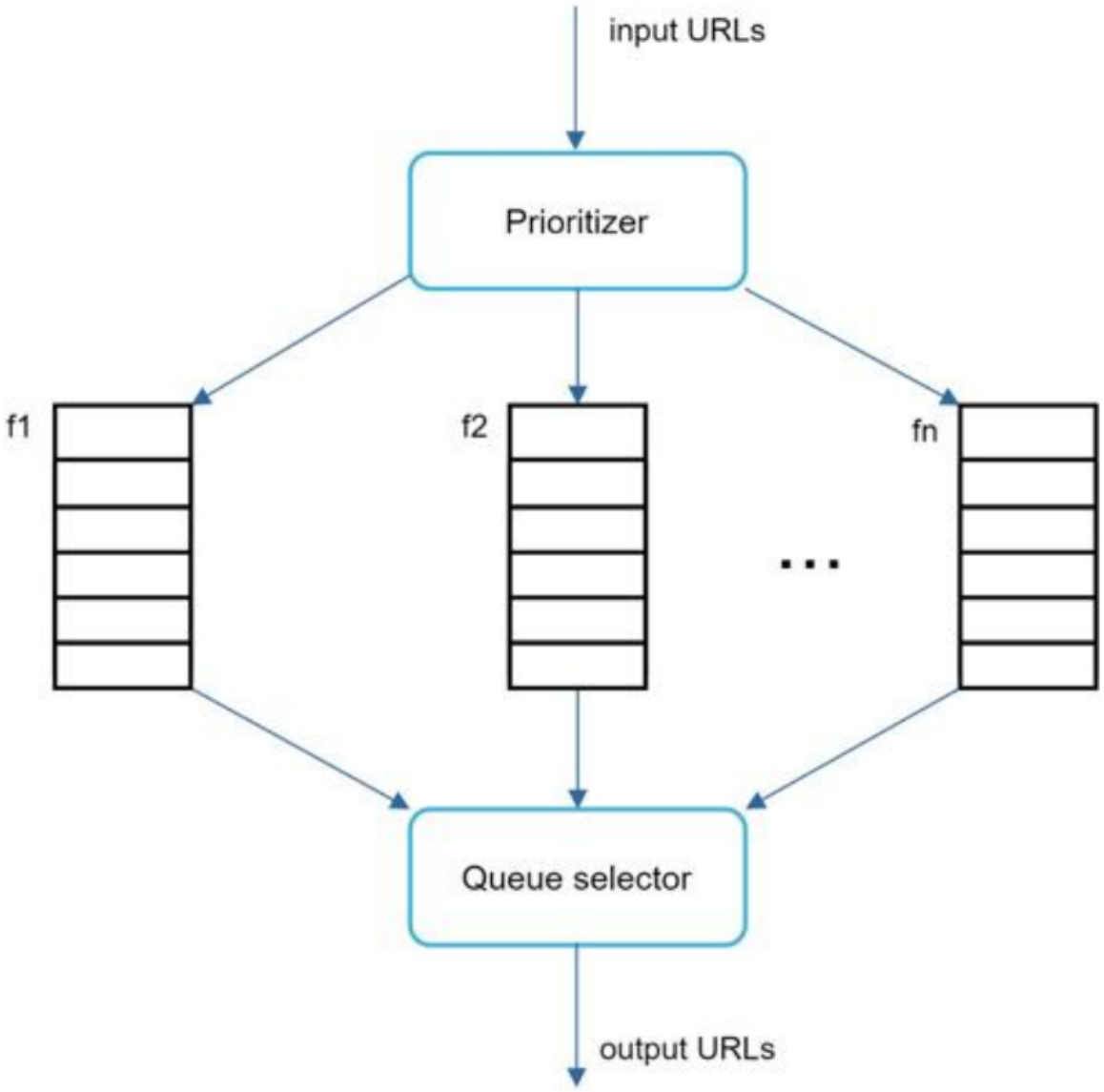


Figure 9-7

- من خلال الصورة أعلاه، يظهر أن ال Prioritizer سيقوم بأخذ ال URL ومن ثم حساب الأولية الخاصة به، وكل Queue من f1 إلى fn لها أولوية مختلفة، الأولية الأعلى منها سيكون لها أولوية أعلى أيضا عند

عمليات الاختيار والسحب من خلال ال Queue Selector، عملية الاختيار تكون عشوائية لكن مع تحيز أكبر وأعلى لل Queue ذات الأولوية الأعلى ^^

والسؤال الذي يطرح نفسه الآن، كيف يمكننا الدمج بين ما تعلمناه في politeness مع ما تعلمناه هنا الآن؟!، والجواب يكمن من خلال أسلوب بسيط وهو تقسيم frontier design إلى two modules وهما ال Front queues والتي ستقوم بدورها بإدارة الأولويات، وال Back queues والتي بدورها ستقوم بإدارة ال politeness..شاهد الصورة:

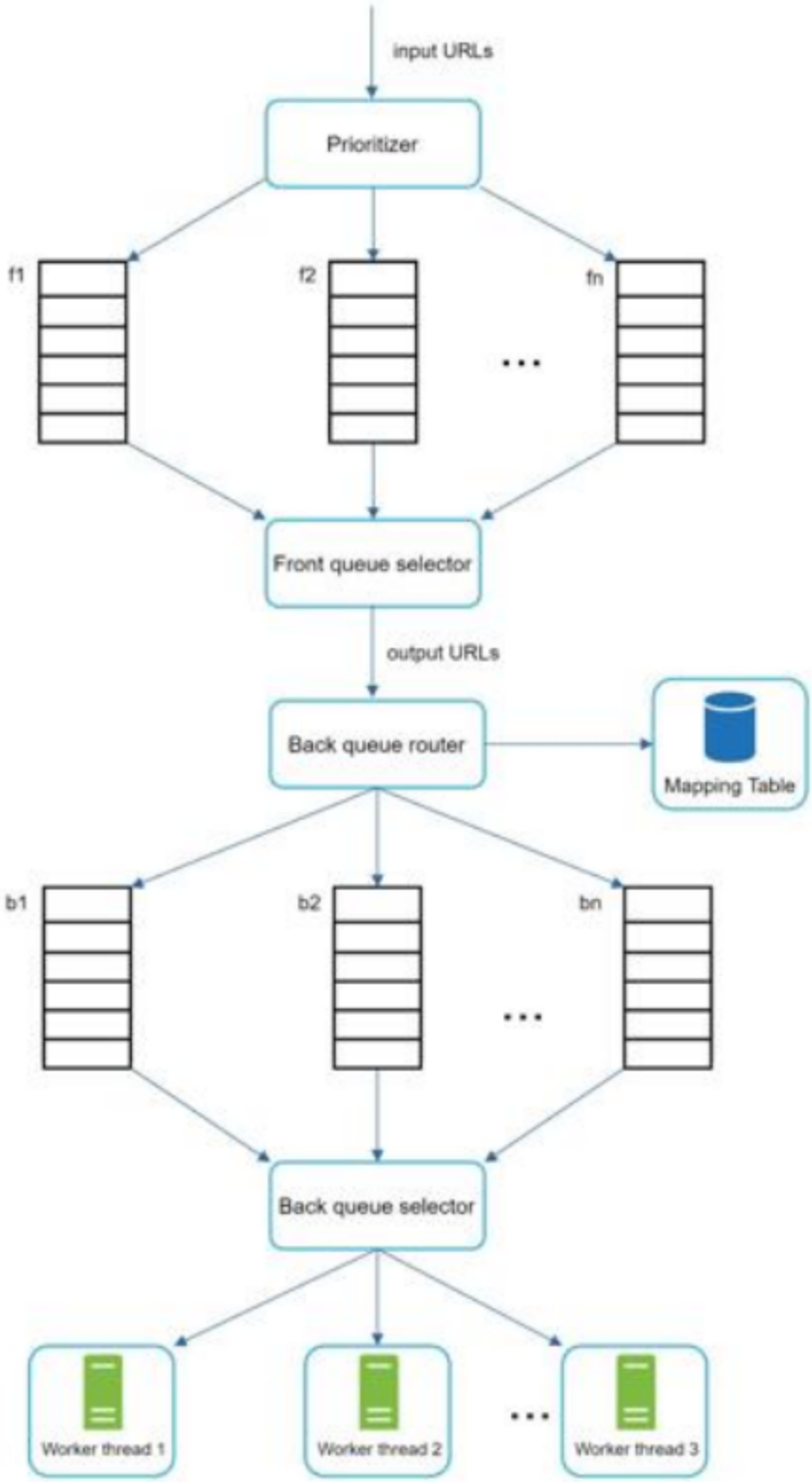

input URLs
Prioritizer
f1
f2
fn
Front queue selector
output URLs
Back queue router
Mapping Table
b1
b2
bn
Back queue selector
Worker thread 1
Worker thread 2
Worker thread 3

Figure 9-8

- Freshness: يجب على ال Web crawler بأخذ نسخة محدثة وبشكل دوري من الصفحات التي تم تحميلها مسبقا، وبكل تأكيد لن نقوم بعمل re-crawler لكل الصفحات مرة واحدة فهذا أمر مكلف وغير منطقي، لكن يمكننا تحسين ذلك من خلال مراقبة ال Web History للصفحات ومن خلال ترتيب أولويات ال re-crawler مجددا ضمن ال queue...

فاصل معرفي: قد تتسائل كيف يمكن معرفة هل حدث هناك تغيير على محتوى الصفحة أم لا، والحقيقة أن هناك أساليب شتى يمكن استخدامها لهذا الغرض، لكن سأذكر هنا لفتة جميلة، وهي أن اعتمادك على عمل request حتى ترى آخر modified date عاد في ال header عملية غير مجدية لأنك ستضطر إلى فعل هذا مع كل ال request، لكن يمكن أن تعتمد على ال age لهذا الغرض، وذلك من خلال اتباع ال Poisson distribution، فمن خلال إضافة توزيع بواسون إلى معادلة تقدير العمر يمكنك توقع العمر الخاص بالصفحة ومنها عمل re-crawler (مجرد لفتة لإحدى الأفكار)

- HTML Downloader: هذا الجزء هو المسؤول عن تحميل الصفحات من الانترنت من خلال بروتوكول ال HTTP، وهذا يقودنا لجزئية مهمة يجب أن يتعامل معها ال Crawler بشكل صحيح، وهي ال Robots Exclusion Protocol، والذي تجده في المواقع الإلكترونية من خلال الملف Robots.txt...
  - Robots.txt: يمثل هذا الملف البروتوكول القياسي للتواصل بين المواقع الإلكترونية وال Crawler، فمن خلاله يتم وضع الصفحات التي يسمح لل

Crawler بتحميلها والصفحات التي لا يسمح له بتحميلها، وبهذا فإن أول خطوة عند تحميلك لصفحات موقع ما، هي التحقق من وجود ال Robots.txt وقراءة محتوياته، وحتى لا نقوم بهذه العملية مرات كثيرة، نقوم بعمل cache لهذا الملف ^^

- أيضا من النقاط المهمة التي يجب أن تراعيها هي ال Performance، فأنت قد تتعامل مع مئات الملايين من الصفحات، لذلك هناك بعض النصائح التي يمكنك اتباعها لهذا الغرض:
  - Distributed crawl: بكل بساطة قم بفصل ال crawl job على أكثر من server، كل server فيه ال thread الخاصة به...كل server من ال download server يصله جزء من ال url ليتعامل معه...
  - Cache DNS Resolver: قم بعمل Cache لل IP address الخاص بالمواقع التي قمت بزيارتها، وذلك حتى توفر الوقت وتحسن الأداء، فهذه العملية تكلف 10ms إلى 200ms تقريبا مع كل عملية تحويل، وهذا سيقوم بعمل Block لل process...لذلك عملية ال cache هنا مفيدة، ويمكن كتابة cron job لغايات تحديث هذا ال cache...
  - Locality: يمكنك أيضا تحسين الأداء من خلال توزيع ال crawl server في أكثر من موقع (geographically) بحيث تكون قريبة من ال host المطلوب...
  - Short timeout: يفضل أن تقوم بتعريف أقصى وقت يمكن لل crawl أن ينتظره قبل أن يغلق، فبعض الصفحات قد تكون بطيئة أو

هناك مشكلة ما على ال host، فبدلا من الانتظار وعمل block للمهمات التي بعدها، نقوم بإغلاق ال task الحالية...وهذه الحركة مفيدة جدا...ويمكن إدارة الصفحات التي لم يتم تحميلها لاحقا...

- Robustness: بجانب اهتمامنا بالأداء فيجب أن نهتم بالمتانة الخاصة بال Crawler، ومن النقاط التي يمكنك الاهتمام بها هي:
  - Consistent hashing: استخدام ال Consistent hashing لتوزيع العمل والأحمال على ال downloaders، فيمكن من خلاله إضافة وحذف ال downloader server بسهولة
  - Save crawl states and data: يجب حفظ البيانات وال state الخاصة بال crawl حتى إذا ما حصل خطأ ما أن يتم تداركه والرجوع إلى آخر State وبيانات تم حفظها...
  - Exception handling: يجب على النظام التعامل مع الأخطاء التي تطرأ أثناء ال crawling مع الصفحات دون أن يتعطل النظام
  - Data validation: يجب أن التحقق من البيانات والحرص على توافقها مع ما نريد حفظه منعا لحدوث أي خطأ في النظام
- Extensibility: القدرة على إضافة Module جديدة تتعامل مع أنواع مختلفة من البيانات نقطة مهمة جدا في ال Web Crawler، وهذا يتم ببساطة من خلال إضافة ال module على ال "content seen؟"...شاهد الصورة أدناه:

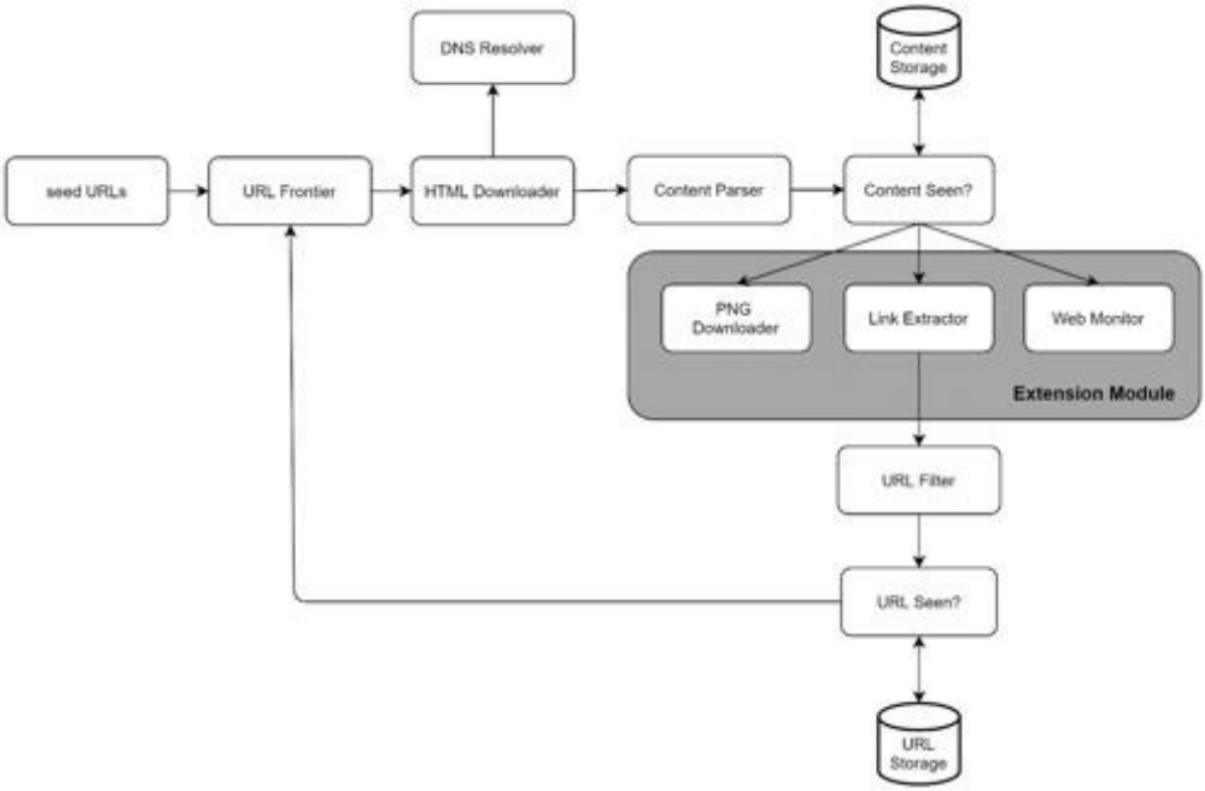


Figure 9-10

- Detect and avoid problematic content: هذه النقطة مهمة جدا أثناء تعاملك وبناء لل Web Crawler، البيانات قد تتشابه أو تتكرر في الكثير من الصفحات وبما نسبته تقارب 29%، لذلك هناك بعض الإجراءات التي يمكنك اتخاذها لهذا الغرض وهي:
  - Redundant content: من خلال استخدام ال Hash للمحتوى ومقارنة ال Hash يمكنك منع التكرار المتماثل للمحتوى
  - Spider traps: يجب عليك الحرص أن لا تدخل بدوران غير منتهي عند عمل ال Crawler، وهذا يحدث للعديد من الأسباب، وحل هذه المشكلة يكون

في معرفة المشكلة التي سببت هذا الدوران اللامنتهي، ومن الحلول أن تقوم بوضع حد معين لطول ال URL بحيث لا يتجاوزه ال Web Crawler، بالإضافة إلى كتابة مجموعة من الحالات الخاصة يتناسب مع موقع معين ونحو ذلك أو استثنائها من ال Crawler...إلى آخره

- Data noise: أي صفحة لا تجد فيها بيانات حقيقة أو فيها مجرد Script ونحو ذلك، عليك الحذر منها وتجاهلها ما استطعت ذلك، فهي قد تسبب لك مشاكل أمنية أو ستضيف إليك محتوى غير مهم (فارغ) ونحو ذلك...

ملاحظة مهمة في الختام

ما ذكر أعلاه يشمل النقاط الأساسية لهذا الموضوع، لكن هناك نقاط أخرى قد تواجهها أثناء عملك وتحتاج منك إلى حل، مثل وجود مواقع تقوم بإنشاء الروابط Dynamic أو بناء على حدث معين يقوم به المستخدم...إلى آخره، لذلك فكر كيف تحل هذه المشكلة، وكيف يمكنك إضافة Spam Filter ونحو ذلك على مخطط العمل الخاص بك...

فائدة

من الجميل أن يدرك قارئ القرآن القاصد لمعانيه والباحث عن كنوزه الراغب في العيش به؛ قصص النزول وتعامل الصحابة معها أو تعامل التابعين معها بما علمها الصحابة لهم أو نقلوا لهم من أحداث ومعاني، إنها تعطينا معاني واقعية نعيشها بين الصعاب والآلام والابتلاءات، بين الأفراح والنعيم والرغبة بنيل الجنات...إنها مفتاح لحياة واقعية، يملؤها الأمل في رضا الله -سبحانه وتعالى-، يملؤها الأمل في أن نكون كما أراد الله -سبحانه وتعالى- لنا...

DESIGN A NOTIFICATION SYSTEM

تعد ال Notification واحدة من التطبيقات المهمة والمستخدمة بكثرة في حياتنا العملية، والتي لا بد لنا من استخدامها أو مشاهدتها أو المرور عليها أو برمجتها وتطبيقها!، كما أن ال Notification ليست شكلا واحدا، بل عدة أشكال، فمنها ما يكون على شكل Push Notification ومنها ما يكون على شكل Email، ومنها ما يكون على شكل رسالة SMS...، وجميعها تستخدم لإشعار المستخدم بوجود أمر ما مهم أو خبر ما تريد إخباره به، ويشمل هذا المشاكل الخطيرة والمنتجات الجديدة والفعاليات ونحو ذلك...

بكل تأكيد بناء نظام يقوم بإرسال الملايين من ال Notification يوميا ليس أمرا سهلا!، ويحتاج إلى فهم عميق لحدود ونطاق ال Notification...، وهذا ما سنتحدث عنه بإذن الله تعالى في هذا الفصل، وكما هي العادة...سنبدأ بوضع مجموعة من الأسئلة التي ستحدد نطاق النظام الذي سنقوم بشرحه...

نماذج من الأسئلة التي يجب أن تسألها لنفسك عند البدء بتصميم نظام الإشعارات:

- ما هي الأنواع / الأشكال التي سيتم إرسال الإشعارات من خلالها؟ هل هي SMS؟ أم Mobile Push أم Email؟ أم جميعهم سيكونون موجودون في الخدمة؟...وسنفترض بدورنا أن جميع هذه الأشكال مطلوبة ^^

- ما هي الأجهزة التي يجب أن تدعم هذه الإشعارات؟ وسنفترض أنها أجهزة ال Android, IOS وال Desktop/laptop

- ما هي طبيعة نظام الإشعارات؟ هل هو real-time system؟، وما هو النوع المناسب تطبيقه من أنواع ال real-time system؟
  سنفترض أن النظام real-time system، والنوع المناسب لنا هو soft real-time

- بناء على ماذا يمكن أن تنطلق (Trigger) الإشعارات؟
  سنفترض أنها من خلال ال Client Application أو من خلال جدولة مسبقة على ال server

- هل يمكن للمستخدم وقف الإشعارات أو تفعيلها (opt-out) بناء على قراره الشخصي؟ سنفترض أن الإجابة نعم

- ما هي كمية الإشعارات المتوقع إرسالها يوميا؟
  سنفترض أنها 10 مليون إشعار للهاتف المحمول، و 5 مليون إشعار من خلال البريد الإلكتروني، و 1 مليون إشعار من خلال رسائل ال SMS

الآن، بعد تحديد متطلبات المشروع، يمكننا أن ننظر إلى ال High Level Design، والذي سيبنى من خلال ثلاثة محاور وهي:

1. Different types of notifications
2. Contact info gathering flow
3. Notification sending/receiving flow

المحور الأول: Different types of notifications:

في هذا المحور علينا النظر إلى طريقة التعامل أو إرسال الإشعارات من خلال الأنظمة المختلفة، وهي ال Android, IOS, Email and SMS...ولكن الصورة العامة كما يلي:

- IOS push notification: لإرسال إشعار إلى أجهزة ال IOS، هناك ثلاثة مكونات أساسية نحتاجها، وهي:
  - Provider: ويمثل هذا المكون المكان الذي سيقوم ببناء الإشعار وإرساله إلى ال APNs، هذا الإشعار الذي تم بناءه مكون من عدة أجزاء وهي:
    - Device Token: وهو ال Unique Identifier المخصص لإرسال الإشعارات
    - Payload: وهذا يمثل ال JSON الذي يحتوي على البيانات الخاصة بالإشعار
  - APNS: هي اختصار ل Apple Push Notification Service، وهي Service مقدمة من Apple لتستقبل الإشعارات المراد إرسالها ومن ثم إطلاقها إلى أجهزة ال IOS
  - IOS: وهي الأجهزة الخاصة بال Client والتي ستستقبل الإشعارات

- Android push notification: المكونات هنا شبيهة بال ios، لكن الفرق أن المكون الوسيط هو ال FCM بدلا من ال APNS، وال FCM هي اختصار ل Firebase Cloud Messaging...
- SMS: نفس الأفكار السابقة، لكن الوسيط هنا سيكون Third party تقدم هذه الخدمة، مثل شركة Twilio...
- Email: الشركات تقوم بإرسال الإيميلات بعدة طرق، منها من يقوم بإرسال البريد الإلكتروني بنفسه ويقوم ببناء هذه ال Service، وشركات أخرى تقوم بالتعاقد مع Third Party يقدم هذه الخدمة مثل Mailchimp، والسبب في استخدام مثل هذه ال Service الحصول على فرصة أكبر وأفضل من الإرسالات الناجحة والقدرة على تحليل البيانات الخاصة بالإيميلات المرسلة...

بهذا، يصبح لدينا شكل التصميم كالآتي:

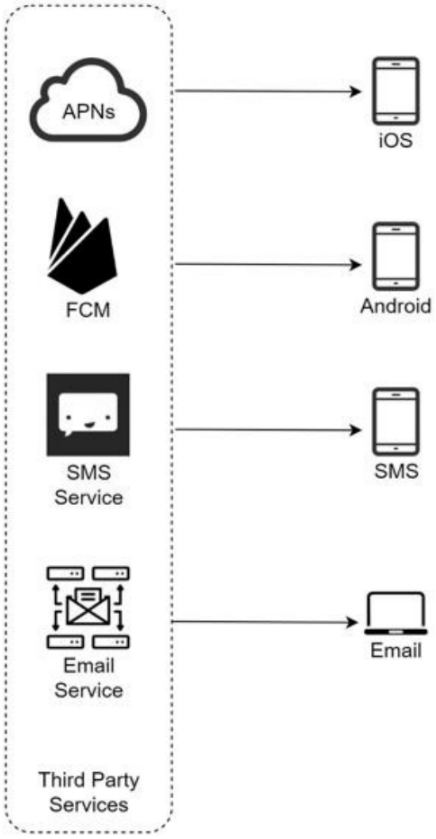


Figure 10-6

المحور الثاني: Contact info gathering flow:

عند رغبتك بإرسال أي إشعار فإنك يجب أن تملك الحد الأدنى من المعلومات التي تسمح لك باستهداف فئة معينة ومن ثم إرسال الإشعارات إليها...، وهذا لا يتم إلا من خلال عملية جمع البيانات الخاصة بالمستخدمين المراد الوصول إليهم، ومن المعلومات التي يمكن جمعها ال mobile device token ورقم الهاتف وعنوان البريد الإلكتروني...، عملية الجمع هذه يمكن أن تتم من خلال العديد من الطرق منها على سبيل الذكر لا الحصر عند تحميل تطبيق معين أو عند التسجيل في الموقع يمكننا إرسال هذه المعلومات ومن ثم حفظها لدينا في قاعدة البيانات

الخاصة بنا، شاهد هذا المثال:

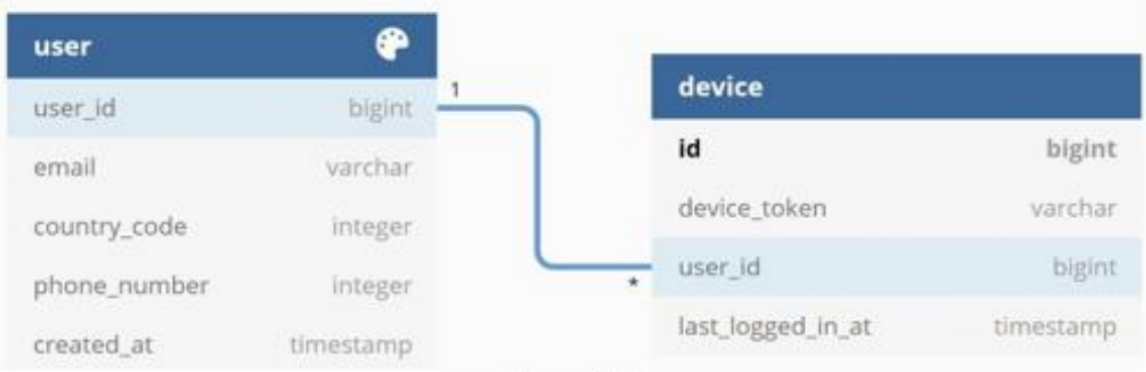


Figure 10-8

في الصورة السابقة قمنا بحفظ المعلومات داخل قاعدة البيانات، لدينا جدولين، الجدول الأول هو الجدول الخاص بحفظ معلومات المستخدمين، والجدول الثاني هو المسؤول عن حفظ الأجهزة الخاصة بالمستخدمين، رقم الهاتف والبريد الإلكتروني تم وضعهم داخل جدول المستخدمين، في حين تم وضع قائمة الأجهزة في جدول واحد وذلك لإمكانية وجود أكثر من جهاز لدى المستخدم الواحد، وهذا يتيح لنا إرسال الإشعارات إلى جميع أجهزة المستخدم عند الحاجة...

المحور الثالث والأخير لدينا من ال High level design للإشعارات هو ال Notification sending/receiving flow:

في هذا المحور سنتحدث عن مخطط سير العمل لإرسال الإشعار من المرحلة الأولى وحتى النهاية، مجموعا فيه مجموعة الأفكار والمحاور التي تم طرحها سابقا، وسنبدأ أولا بالشكل التقليدي أو الفكرة الأولى لتنفيذ نظام الإشعارات والتي ستخطر على بالك، ثم سنقوم بطرح المشاكل الخاصة بها، والآن، شاهد هذه الصورة:

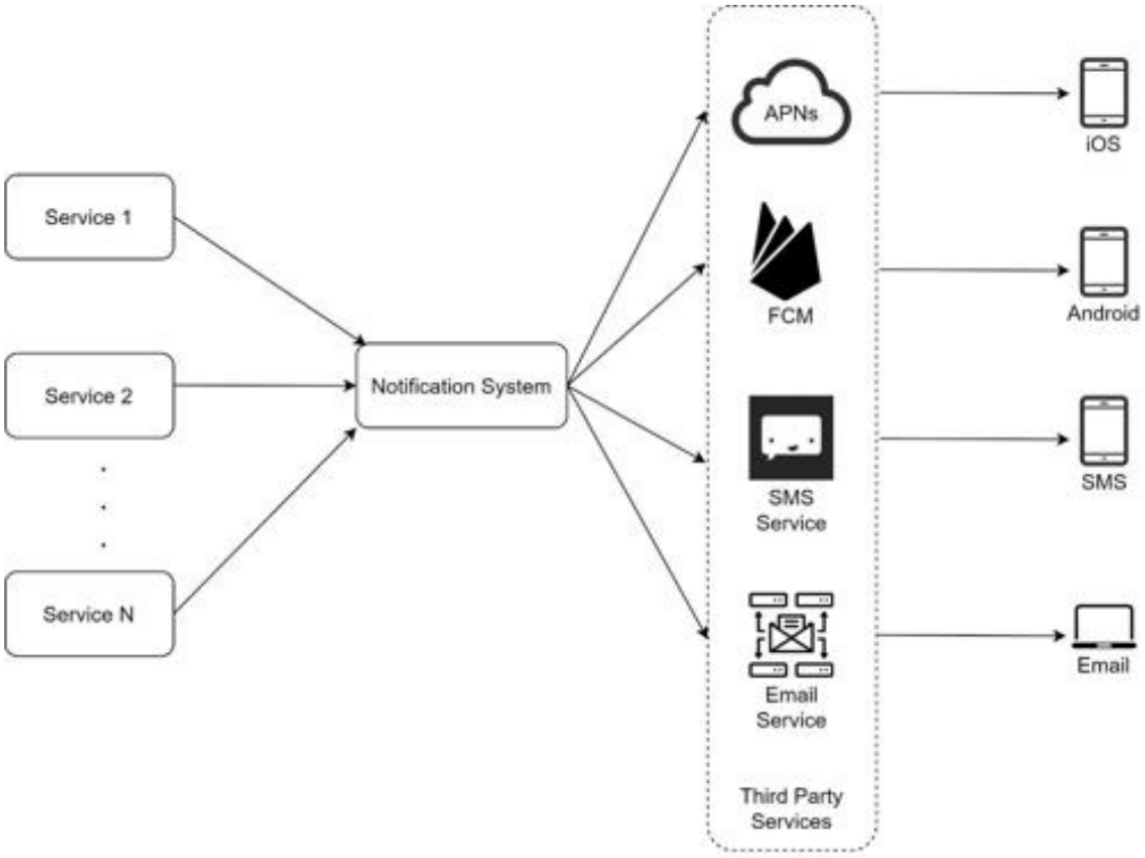


Figure 10-9

في هذه الصورة لدينا عدة مكونات، بحيث يمثل المكون الأول ال Service التي ستقوم بعمل trigger event لإرسال ال notification، وهذا يمكن أن يكون cron job أو micro-service أو distributed system...، والمكون الثاني يمثل المكان الذي سيقوم باستقبال ال event من ال service من خلال API بقدمها، ومن ثم القيام ببناء ال payload لل third party، والمكون الثالث والرابع تم شرحها سابقا...

كما تلاحظ لدينا ثلاثة مشاكل رئيسية هنا، وهي:

1. Single point of failure (SPOF)

2. Hard to scale

3. Performance bottleneck

ولتحسين هذا النموذج، يمكننا اقتراح التحسين التالي: (لاحظ أن فكرة الكتاب وما نتحدث عنه هنا هو بناء نظام يمكنه خدمة الملايين من المستخدمين أو ال requests، فدع ذلك في مخيلتك دوما)

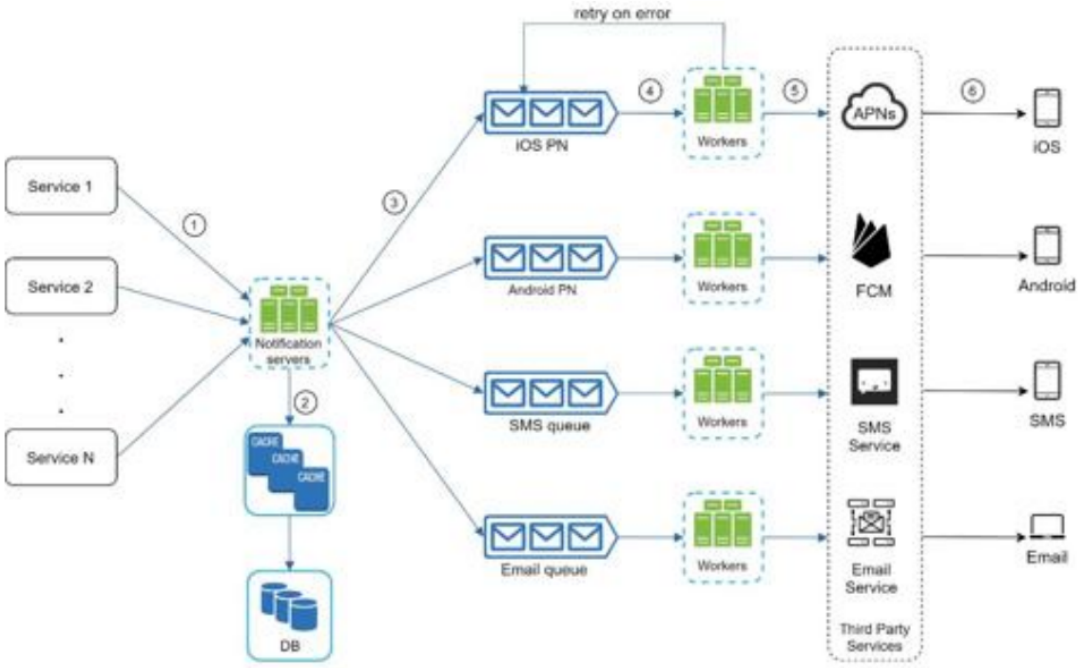


Figure 10-10

تتلخص عملية التحسين من خلال ثلاثة نقاط أساسية وهي:

1. نقل قاعدة البيانات وال cache خارج ال notification server

2. إضافة أكثر من notification server وذلك من خلال القدرة على بناء horizontal scale
3. إضافة Queue لتغطي ال notification component الموجودة

من خلال النقاط السابقة يمكننا فهم الصورة السابقة، فال Service من 1 إلى N تمثل مجموعة ال service التي ستقوم بتقديم ال event / الإشعارات إلى ال notification servers ليتم إرسالها، ثم ال Notification servers ستقوم بتقديم ال API إلى ال service لتيم إرسال الإشعارات من خلالها، ويكون الوصول إليها أو تطبيقها من خلال عملية مؤمنة حتى لا تقع ضحية ال SPAM، كما أنه يقوم بالتحقق من البيانات مثل ال email, phone number، ويقوم بجلب البيانات المطلوبة من قاعدة البيانات أو الكاش ليتم بناء الإشعار، ثم بعد كل ذلك سيقوم بوضع هذا الإشعار داخل ال Queue ليتم معالجة هذه الإشعارات داخل ال Queue بشكل parallel، وال Message queue أهم وظيفة لها في التصميم الخاص بنا هو إزالة الاعتمادية الموجودة بين العناصر أو المكونات المختلفة في التصميم، وهنا يأتي دور ال worker ليقوم بسحب الإشعارات من ال message queue ومن ثم إرسالها إلى ال Third party service...

الآن، أصبحت الأمور واضحة ومفهومة بشكلها العام، لذلك، علينا أن ندخل إلى بعض التفاصيل بشكل أعمق وأدق، حتى يصبح لدينا تصميم فعال لهذا النظام...
أثناء حديثنا عن ال High Level design قمنا بتغطية أنواع الإشعارات التي يمكن إرسالها وكيفية جمع المعلومات واستقبال وإرسال الإشعارات ونحو ذلك، والآن سنتطرق إلى:

1. Reliability: قدرة النظام على الحفاظ على مصداقيته وموثوقيته في تنفيذ المهام وإيصالها بدون فقدان أي منها وفي الوقت المحدد أمر مهم جدا لأي نظام، وهذا يُلزمنا أن نسأل أنفسنا الآتي:
   a. كيف يمكن أن تحمي نفسك من أي خسارة في البيانات؟، فواحدة من أهم المتطلبات لأي نظام إشعارات هو ضمان عدم خسارة أي إشعار، يمكن أن تتأخر لسبب ما أو يعاد ترتيب أولويتها، لكن، لا يمكن تقبل خسارة إشعار ما!، لذلك يكون الحل هنا من خلال آلية أو عملية التكرار (محاولة إعادة العملية حتى يتم ضمن نجاحها) وتسمى هذه العملية ب Retry Mechanism.
   b. هل سيستقبل العملاء الإشعار لمرة واحدة بالضبط؟، رغم بساطة السؤال وتوقع الإجابة بأنها نعم!، إلا أن الإجابة الحقيقية هي لا!، إن نظام الإشعارات وإن كان يرسل الإشعارات لمرة واحدة فقط معظم الأحيان، إلا أن هناك نسبة خطأ موجودة في الأنظمة ال distributed، لذلك يجب أن يكون هناك آلية تحد أو تقلل من كمية الإشعارات التي يمكن أن تتكرر، ومن الطرق لذلك التأكد من الإشعار قبل إرساله هل شوهد من قبل أم لا...، هناك ثلاثة أنواع مختلفة من طرق التسليم وهي ,at-most-once at-least-once, and exactly-once، الخيار الأول الثاني يمكن تطبيقهما، لكن الخيار الثالث (فقط مرة واحدة لا أقل ولا أكثر) لا يمكن تطبيقه إلا بوجود ضمانات على كافة المستويات والمكونات، وهذا شيء غير ممكن عمليا أو ذو تكلفة عالية غير قابلة للتطبيق على أقل تقدير!

2. Additional component: لقد تحدثنا سابقا عن بعض المكونات التي نحتاجها أثناء عملنا على نظام الإشعارات مثل الطريقة الخاصة بجمع البيانات وإرسال الإشعارات ونحو ذلك، لكن هناك مكونات أخرى مهمة يجب أن تضعها في الحسبان أثناء عملك على نظام الإشعارات مثل:
   a. Notification Template: عند بناء نظام للإشعارات فإنك ستقوم بإرسال الكثير منها، وبكل تأكيد لن تقوم بإنشاء الإشعار في كل مرة وتقوم بتصميمه في كل مرة!، فعادة ما تكون الإشعارات ذات نمط محدد مسبقا، هذا النمط يتم بناؤه من خلال Template تم إنشاؤه مسبقا، ويتم استبدال القيم المتغيرة بشيء يسمى Replacement Pattern، وهو عبارة عن pattern تقوم أنت بتعريفه ليتم استبداله بقيمة محددة، مثلا: Hello {{{FULL_NAME}}} Mate، في هذا المثال سيتم البحث عن {{{FULL_NAME}}} واستبدالها مثلا ب Anees...، إن عملية إنشاء ال Template عملية مفيدة جدا لتقليل الأخطاء الممكنة وحفظ الوقت في التطوير والتعديل ونحو ذلك
   b. Notification setting: من الأمور المهمة للمستخدمين وجود آلية تمكنه من إيقاف الإشعارات، فالمستخدم قد تصله الكثير من الإشعارات التي قد لا يراها مهمة ويرغب في إيقافها، لذلك تقوم الكثير من التطبيقات والمواقع بوضع لوحة تحكم خاصة بالإشعارات تمكن المستخدمين من إيقاف الإشعارات التي لا يرغبون في استقبالها، أو إيقاف نوع محدد منها، مثلا إيقاف إشعارات الهاتف المحمول وإبقاء إشعارات البريد الإلكتروني، أو إيقاف إشعارات آخر الأخبار وإبقاء إشعارات المشاكل الأمنية ونحو ذلك

c. Rate limiting: من الأمور المهمة أيضا أن تراعي عدد الإشعارات التي يتم إرسالها للشخص وضبطها بعدد محدد لا يزعجه!، فالمستخدم إذا شعر بالانزعاج من كثرة الإشعارات فأقل فعل يمكن أن يفعله هو إغلاق جميع الإشعارات الخاصة بك إن لم يكن حذف التطبيق كاملا!

d. Retry mechanism: يجب أن يكون لديك آلية لإعادة إرسال الإشعارات التي فشلت في الوصول، فلو حدثت مشكلة ما عند ال third party فعليك إضافة هذا الإشعار إلى Fail queue ومن ثم محاولة إرسالها مرة أخرى، فإن لم تنجح العملية يجب عليك إشعار المطورين بحدوث هذه المشكلة، وكل هذا يتم بشكل محوسب ^^

e. Security in push notifications: من الأمور المهمة والخطيرة التي يجب أن تنتبه لها هي موضوع الأمان في الإشعارات، لأن الإشعارات إن لم تكن مبنية بطريقة آمنة فهي طريقة سهلة وخطيرة لانتحال هويتك ومن ثم سرقة منتسبيك!، لذلك تضع مثلا كل من IOS, Android ال Appkey, AppSecret ليتم إرسال secure push notification، ومن الأمور التي يجب أن تراعيها هي (معظم هذه النقاط قد تكون مفعلة ضمنيا عند انتسابك لمزود خدمة خارجي، لكن قمت بطرحها لتدرك طبيعة ما تحتاجه وما يمكن أن تعمل عليه):

i. Data privacy: عليك حماية الخصوصية الخاصة ببيانات المستخدمين لديك، وإرسال البيانات الحساسة لا ينبغي أن يتم إلا من خلال ال secure protocol

ii. Malicious push notifications: يمكن لبعض المخربين أن يقوموا بإرسال أو إدخال برامج خبيثة من خلال الإشعارات، لذلك يجب أن تحمي المستخدمين المنتسبين إليك من خلال نظام آمن يتم فيه بناء ال authentication وال authorization بطريقة جيدة، ومن الطرق المستخدمة لذلك ال digital signatures

iii. Phishing attacks: عمليات التصيد من خلال الإشعارات المزيفة هي عمليات شائعة، فمن المفضل أن يكون لديك آلية لإيصال الرسائل التوعوية إلى المستخدمين المنتسبين لك؛ حول طبيعة الإشعارات الخاصة بك وطريقتها ونوع البيانات التي يمكن أن تطلب أو البيانات التي لا يمكن أن تطلب...مثلا "عزيزي المشترك، اعلم أن شركتنا الموقرة لن تطلب منك نهائيا رقم بطاقتك الإئتمانية، ولن يتم التواصل إلا من خلال بوابتنا الرقمية www.2nees.com، وأي تغيير على هذا الرابط بأي شكل من الأشكال هي عملية انتحال شخصية قد تعرضك للخطر"

f. Monitor queued notifications: مراقبة ال Queue أمر مفيد جدا للعديد من الأسباب، واحدة منها إذا تواجد عدد كبير من الإشعارات المراد إرسالها والتي تخرج عملية الإرسال عن السرعة المعتادة فيمكننا زيادة عدد ال Service worker...، وهذا بدوره يمنع أو يقلل أي delay محتمل للإشعارات...

g. Events tracking: عملية متابعة ما يحدث للإشعارات عند وصولها للمستخدم أمر مهم وإلزامي لنظام الإشعارات، وتعود أهمية ذلك لمعرفة وتتبع سلوك

المستخدم، ومنها توقع مستوى رضاه ومستوى الإشعارات التي لديك ونحو ذلك...، ومن ال event المهم تتبعها ال open rate, click rate وال unsubscribe...

3. Updated design: الآن...بعد كل المكونات والأجزاء التي تحدثنا عنها...يمكننا النظر إلى الشكل النهائي:

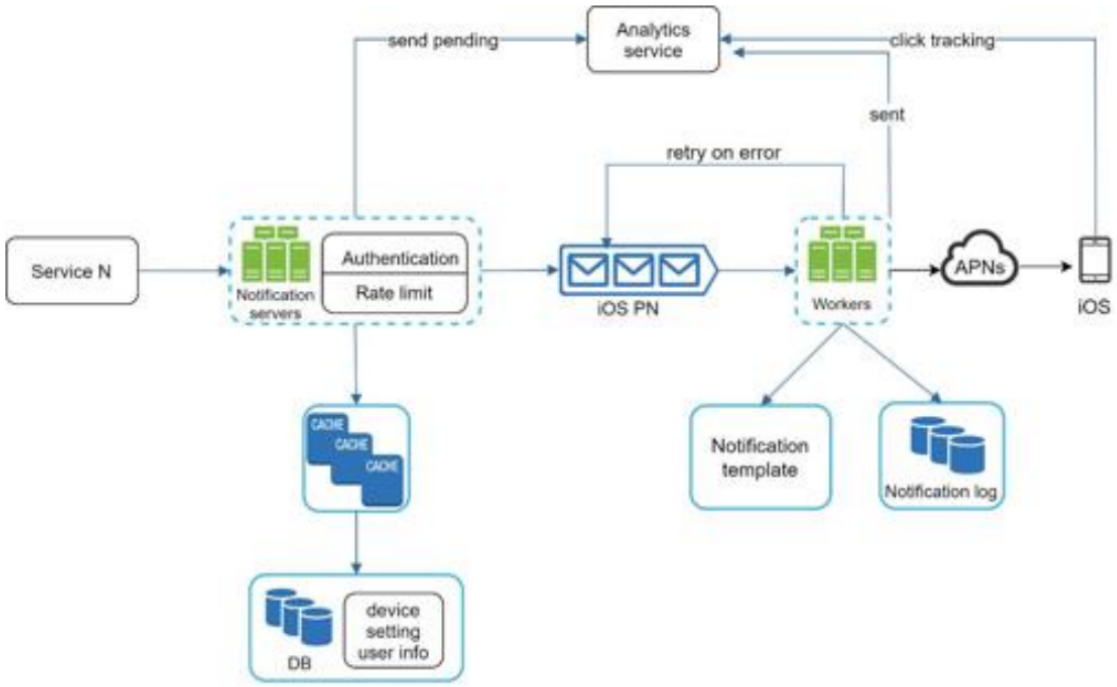


Figure 10-14

في التصميم المرفق أعلاه تمت إضافة الأجزاء التي تحدثنا عنها، ولتبسيط الصورة لنقم بتجزئتها:

a. تمت إضافة ال Authentication وال rate-limiting إلى ال notification servers
b. تمت إضافة ال retry mechanism، بحيث سيتم إعادة الإشعار الذي فضل إرساله لل Queue، وطبعا سيكون هناك رقم معرف مسبقا بعدد المرات الأعلى الذي يسمح فيه لإشعار فاشل بالعودة لل Queue...
c. تمت إضافة ال notification template ليتعامل معها ال workers
d. تمت إضافة ال tracking system للنظام.

فائدة

إن سيرة الحبيب المصطفى -صلى الله عليه وسلم-؛ هي أعظم قصة نجاح نالها بشر على هذه الأرض، ولأن الناس مولعون بتتبع أخبار الناجحين والمميزين على مر العصور، كان ينبغي علينا أن يكون خير البشر -صلى الله عليه وسلم- هو أولى الناس بدراسة تاريخه وسيرته وكل ما اتصف به -عليه الصلاة والسلام-!، وعجبا لقوم يتركون أو يتناسون أو يؤخرون قراءة الهدي النبوي وسيرة المصطفى -عليه الصلاة والسلام- ليقرأوا ويتدارسوا ما قاله أفلاطون وأرسطو وسقراط!، مع أنهم هؤلاء جميعا وغيرهم من الفلاسفة لم يقدموا منهجا علميا رصينا وعمليا ومنهجيا ناجحا كما قدمه سيد الخلق -عليه الصلاة والسلام-!

DESIGN A NEWS FEED SYSTEM

في هذا الفصل من الكتاب سنتحدث عن طريقة تصميم News Feed System، وقبل الشروع في الحديث عن هذا النظام وطريقة تصميمه، يجب أن نعرف ما هو هذا النظام، ويمكن القول بأنه -مما ذكر في صفحة المساعدة للفيسبوك بتصرف-: "أي List موجودة لديك في وسط الصفحة الرئيسية وتحتوي على Stories يتم تحديثها بشكل مستمر، وهذا يشمل ال status update والصور والمنشورات والفيديوهات والاعجابات ونحو ذلك"، هذا التعريف سيعطيك انطباعا عن أهمية هذه الجزئية في حياتك العملية وفي طريقة تنفيذ التصاميم الخاصة بمشاريعك الحالية، لهذا يعد هذا السؤال "قم بتصميم Facebook news feed, instagram feed, twitter timeline" واحدة من أشهر أسئلة المقابلات...

وكما اعتدنا، لنبدأ بجمع المعلومات الخاصة بهذا النظام لفهم حدود المشكلة ونطاق التصميم...

- هل سيكون هذا النظام خاص بتطبيقات الويب أو الهاتف المحمول أن لكليهما؟، وليكن الجواب أن هذا النظام سيخدم الويب والهاتف المحمول
- ما هي أهم المزايا التي ترغب بوجودها في هذا النظام؟
وليكن أن يتمكن المستخدم من نشر منشوراته مع إمكانية مشاهدة هذه المنشورات من قبل الأصدقاء من خلال صفحة ال news feed page
- كيف سيتم ترتيب هذه المنشورات أو الأحداث؟ هل سيتم ترتيبها بناء من الأحدث إلى الأقدم؟ أم سيتم ترتيبها بناء على المنشور صاحب النقاط الأعلى (مثلا المنشور

الذي يحتوي صورة له أولوية أعلى من المنشور الذي يحتوي نص فقط أو المنشور الذي لديه إعجابات أكثر له أولوية أكبر)، أم سيتم ترتيبها بناء على درجة القرابة الخاصة بالأصدقاء (مثلا الأصدقاء إذا كانوا من العائلة لهم أولوية أعلى...) وليكن الترتيب من الأحدث للأقدم حتى يكون الأمر أسهل...

- ما هو أكبر عدد من الأصدقاء الذي يمكن أن يمتلكه أي مستخدم؟
  وليكن 5000
- كم عدد المستخدمين النشطين يوميا؟
  وليكن 10 مليون
- هل يمكن أن تحتوي المنشورات على صور أو فيديوهات أم مجرد نصوص فقط؟
  يمكنها أن تحتوي صور وفيديوهات بالإضافة للنصوص...

بناء على الأسئلة السابقة أصبح لدينا تخيل عن طبيعة النظام الذي نرغب ببنائه، والآن يمكننا الانتقال للبدء بتصميم هذا النظام بشكله العام...

يمكننا أن نقسم هذا النظام لجزئين، هما:

1. Feed Publishing: وهو الجزء الذي ينطلق من مرحلة كتابة ونشر المنشور وحفظه في قاعدة البيانات وال Cache، ثم نشره للأصدقاء..، وهذا الجزء لديه Post API ستقوم بإنشاء المنشور، وأخرى Get Api لتقوم بجلب ال Feeds...، والصورة أدناه تمثل التصميم العام لهذا الجزء:

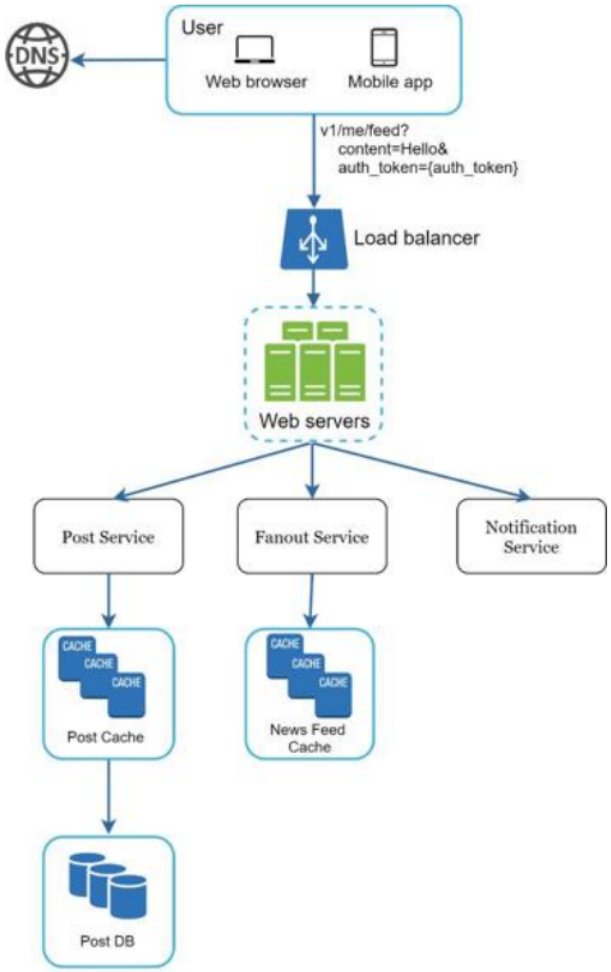


Figure 11-2

لا جديد بالصورة سوى جزئية واحدة وهي internal service:

a. Post Service: وهو المسؤول عن حفظ المنشورات الجديدة داخل الكاش وقواعد البيانات

b. Fanout Service: وهي الخدمة المسؤولة عن إضافة المحتوى الجديد للأصدقاء، وهي عبارة عن عملية تخزين داخل الكاش لتسريع عملية الاسترجاع للمحتوى

c. Notification Service: عند وجود محتوى جديد، يمكننا إشعار الأصدقاء بوجود هذا المحتوى...

2. Newsfeed building: وهو الجزء الخاص ببناء ال news feed الخاص بك من مجموعة الأصدقاء التي لديك، ولتبسيط الفكرة قلنا أن هذا سيكون من خلال الترتيب الزمني (الأحدث للأقدم مثلا...)، وهذا التصميم الخاص بها:

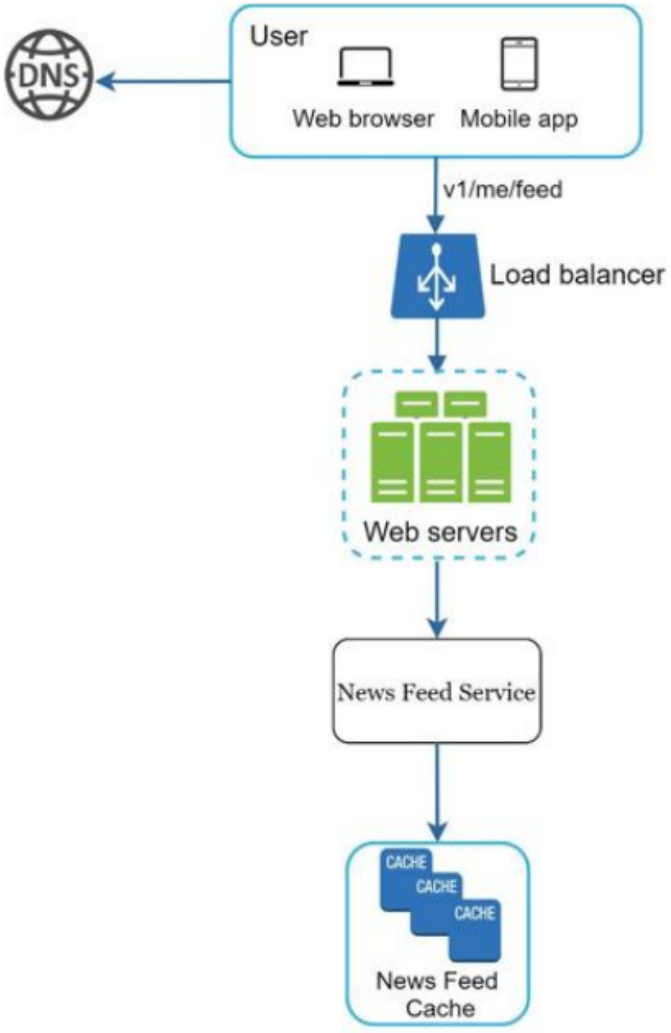


Figure 11-3

والجديد في هذا التصميم لهذا الجزء:

- Newsfeed Service: وهي الخدمة المسؤولة عن جلب ال Newsfeed من ال Cache

- Newsfeed cache: وتقوم بتخزين ال news feed IDs المطلوب استرجاعها داخل ال cache...

والآن لننتقل للتصميم بشكل أعمق وأدق...

1. Feed publishing deep dive: لقد تحدثنا عن التصميم الخاص بهذا الجزء أعلاه، لكن، دعونا ننظر إلى التصميم ومكوناته بشكل أكثر عمقا...شاهد الصورة أدناه:

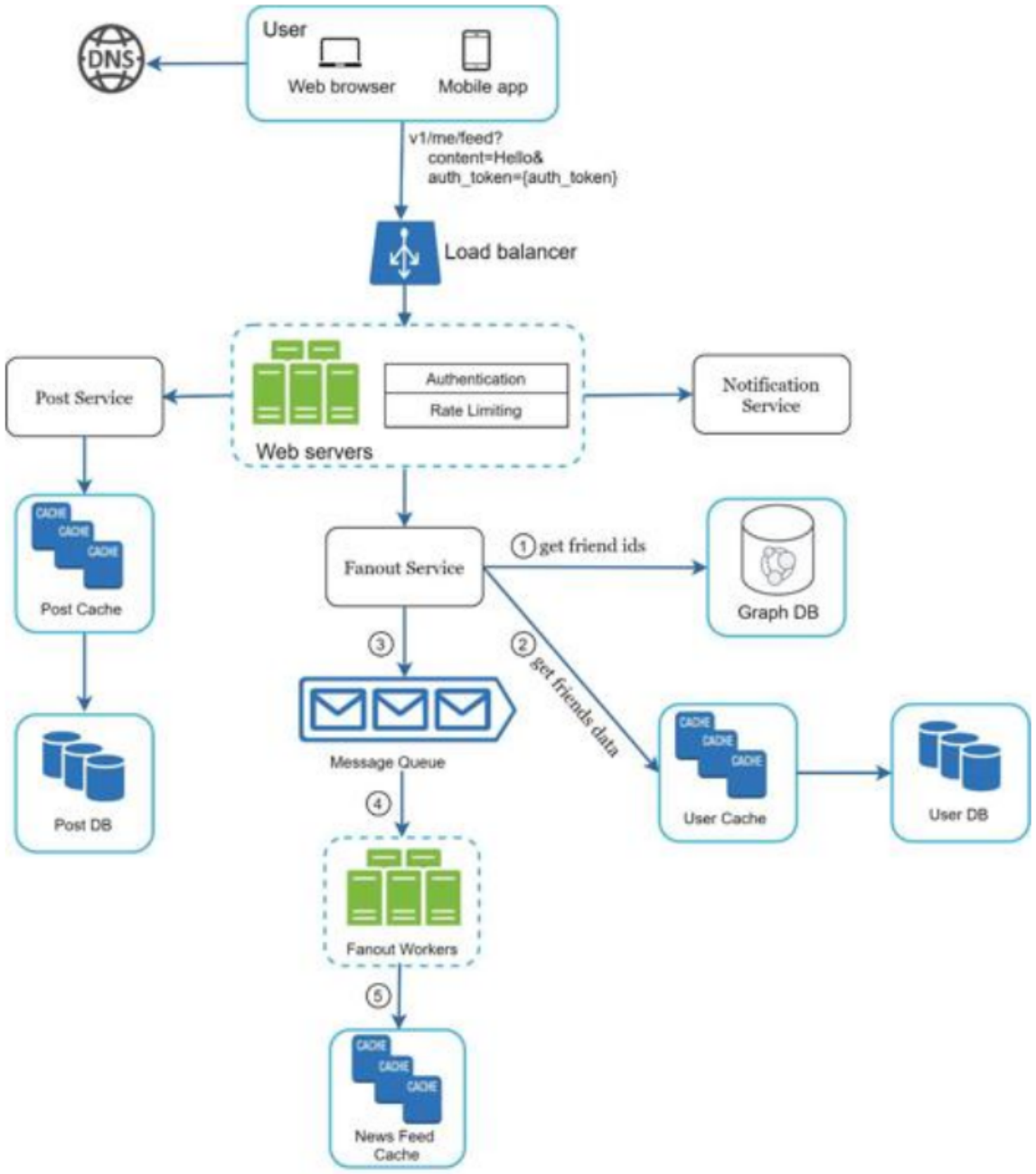


Figure 11-4

تتكون الصورة أعلاه من مكونين رئيسيين لهذا الجزء من النظام، وهما ال Web Servers وال Fanout Service...، لذلك دعونا نشرح بشكل موجز مكونات هذه الصورة:

I. Web servers: تقوم ال Web Servers هنا بأكثر من مهمة، فهي بالرغم من كونها مكان الاتصال بين المستخدمين وبين النظام، إلا أنها أيضا ستقدم لنا خدمة التحقق من المستخدمين وأن كل مستخدم سيقوم بنشر منشور جديد هو مستخدم حقيقي قام بتسجيل الدخول إلى النظام، كما أنها ستقوم بحماية النظام من خلال تنفيذ الشروط على عملية إنشاء المنشورات، مثلا يسمح لكل مستخدم بمنشورين في كل ساعة.... وهذا مهم جدا لمنع ال spam...

II. Fanout service: قد تكون هذه الجزئية هي الأهم من مكونات النظام والتي ينبغي عليك العناية بها، فهذه ال service هي المسؤولة عن إيصال المنشورات إلى جميع الأصدقاء، ولل Fanout model نوعان، وهما:

- fanout on write: ويطلق عليها أيضا Push Model...، في هذا ال model يتم الكتابة مباشرة على friends cache بعد نشر المنشور، ويتميز هذا الأسلوب باعتباره real-time ويمكن أن تظهر المنشورات عند الأصدقاء مباشرة، بالإضافة إلى أن عملية ال fetch ستكون سريعة لأننا قمنا بجلب قائمة الأصدقاء مسبقا أثناء عملية ال Write للمنشور (لا تنسى أننا نتحدث عن ال Fanout service وليس ال client keyboard ^^)، لكن هذا ال model لديه مشاكله التي يجب أن تنتبه لها، وهي إذا كان لهذا الشخص عدد أصدقاء كبير فعملية جلب هذه القائمة وإنشاء feeds لهم جميعا ستكون بطيئة وستستهلك وقتا طويلا، ويطلق عليها "hotkey problem"، كما أن وجود مستخدمين غير

نشطين على النظام سيكون بمثابة إضاعة لل resource على ال feeds لأشخاص لا يقومون بتسجيل الدخول إلا نادرا!

- fanout on read: ويطلق عليه أيضا Pull Model...، في هذا ال model يتم إنشاء ال feeds عندما يقوم المستخدم بالدخول للصفحة (لحظة القراءة يتم عمل generate لل feeds)، بمعنى آخر يتم جلب آخر المنشورات "on-demand"، ,ويتميز هذا الأسلوب أنه يتخلص من مشكلة المستخدمين غير النشطين، ولهذا نحفظ ال Resource بأن تستهلك بلا جدوى، كما أن البيانات لا يتم دفعها لجميع الأصدقاء مرة واحدة، وبهذا سنتخلص من "hotkey problem"....، لكن لهذا الأسلوب أيضا مشكلة مهمة، وهي أن سرعة جلب المنشورات بطيئة لأنها لم تكتب بشكل مسبق كما هو الحال في Push model

بناء على ما سبق ذكره، يمكنك اختيار الأسلوب المناسب للنظام الخاص بك!، لكن يمكنك أيضا الاستفادة من كلا الأسلوبين ودمجها معا للحصول على نتائج مميزة!، وهذا يكون من خلال بناء بعض الفلاتر لجعل منشورات الأشخاص المؤثرين والذين لديهم متابعين كثر أو أصدقاء كثر تقرأ عند الطلب (on-demand)، بينما يتم تطبيق ال push model لغالبية المستخدمين...وبهذا نكون اكتسبنا سرعة قراءة عالية، وبنفس الوقت حافظنا على مواردنا...وقللنا من فجوة المشاكل الممكنة لكل أسلوب...

والآن شاهد التصميم الخاص بال fanout service:

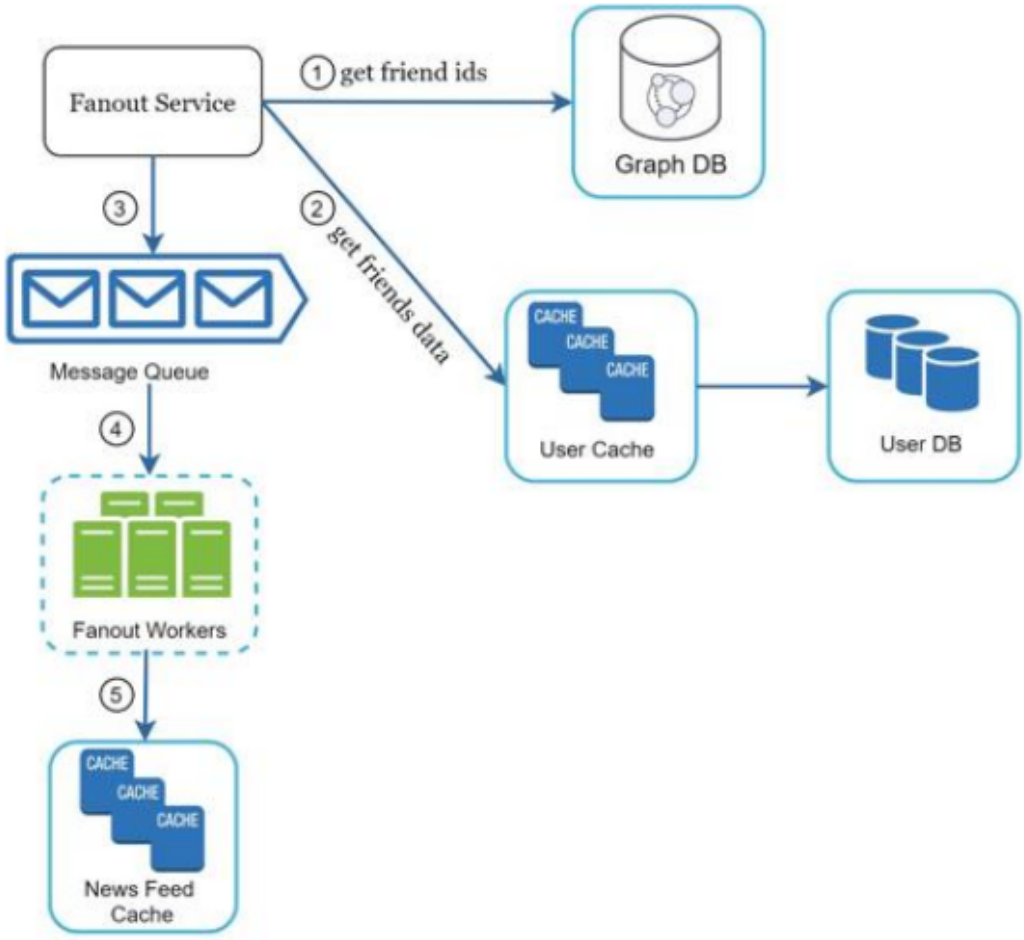


Figure 11-5

هذا التصميم يعمل بالطريقة التالية:

i. يتم جلب friend IDs من graph database، وهذا يتم من خلال ال friend relationship and friend recommendations.

ii. جلب بيانات الأصدقاء من ال user cache، وتطبيق الفلاتر المناسبة على هذه القائمة مثل لو كان هناك صديق وضع علامة "عدم رؤية منشورات س"، أو إذا كان المنشور عام لكل الأصدقاء أو للعائلة فقط ونحو ذلك

iii. إرسال قائمة الأصدقاء وال post id إلى ال message queue

iv. يقوم ال Fanout workers بسحب البيانات من ال message queue ومن ثم يقوم بحفظ feeds داخل ال feed cache، ولأن كمية البيانات يمكن أن تكون كبيرة جدا وال cache لا يتحمل ذلك فإننا نقوم فقط بحفظ ال IDs، كما أن المستخدمين يهتمون بأحدث المنشوارت غالبا فيتم حفظ المنشورات الأحدث في الكاش والمحافظة عليها، وتقل قيمتها مع مرور الوقت لأن المستخدمين قل ما ندر بأن يتصفحوا 1000 منشور مثلا في جلسة واحدة من خلال ال scroll ^^

v. في ال new feed cache يتم حفظ ال feeds على شكل postId, userId

2. Newsfeed retrieval deep dive: الجزء الثاني هو التحدث بشكل أعمق عن عملية جلب ال feeds، شاهد الصورة أدناه:

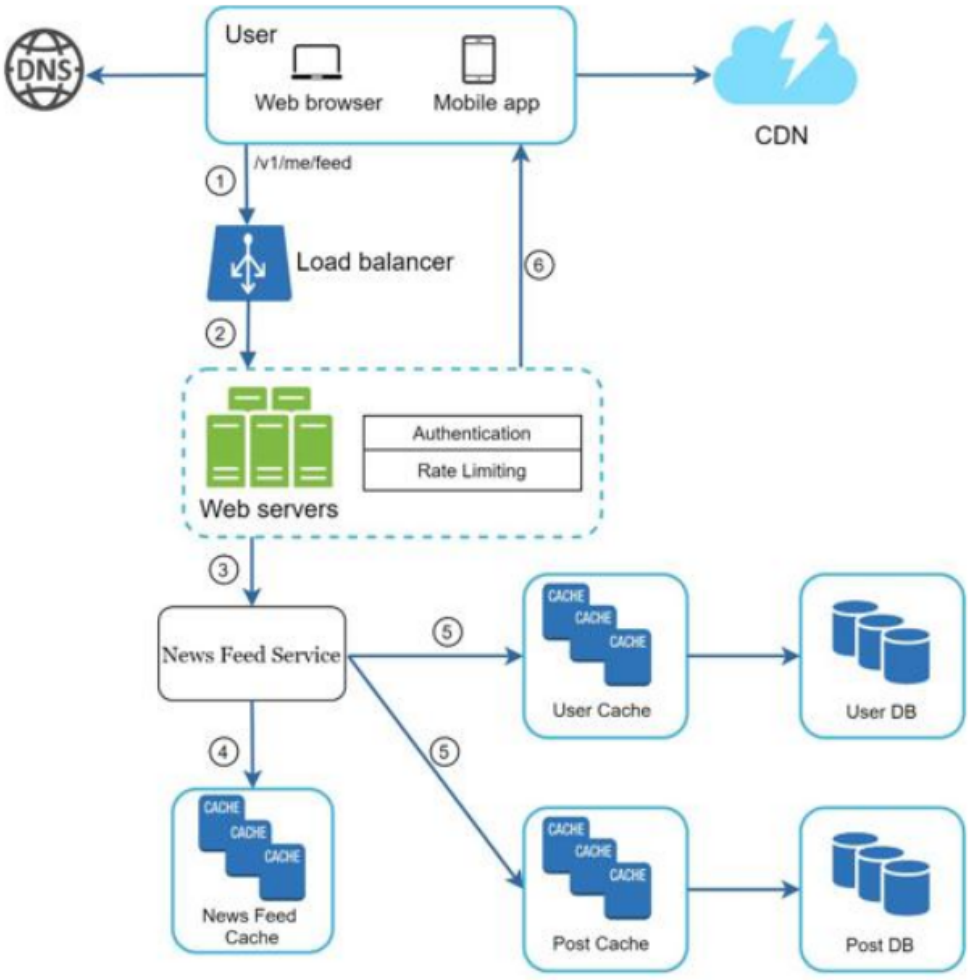


Figure 11-7

a. أولا يقوم المستخدم بعمل request لطلب ال feeds

b. يقوم ال load balancer بتحويل الريكويست إلى السيرفر المناسب

c. تقوم ال web servers بطلب ال feeds من ال service الخاصة باسترجاع ال news feed...

d. ال news feed service تقوم بجلب ال IDs الخاصة بالمنشورات من ال cache
e. لأن المنشورات الخاصة بالأعضاء ليست مجرد IDs سنقوم باستغلال ال IDs لجلب معلومات المستخدم والصور والفيديوهات والنصوص ونحو ذلك وبناء Object متكامل، وهذا يتم من خلال جلب المعلومات من ال user cache وال post cache
f. يتم إرجاع البيانات على شكل JSON لل client.

هذا هو الأمر بكل بساطة ^^، لكن دعونا نتحدث عن ال Cache Architecture هنا، فهو ذو أهمية بالغة في ال news feed...

ال Cache Architecture:

قام Alex Xu بتقسيم ال Cache إلى 5 طبقات وهي:

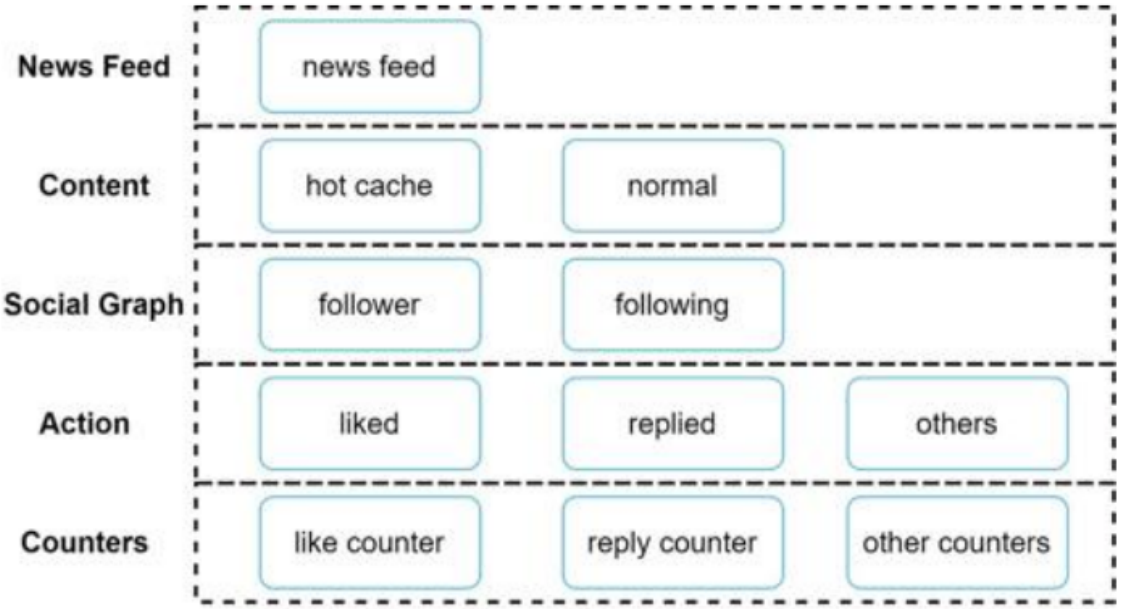


Figure 11-8

- في الطبقة الأولى يتم حفظ ال IDs الخاصة بال news feed
- في الطبقة الثانية يتم حفظ البيانات الخاصة بكل منشور، وهي نوعان، هناك ال hot cache والتي تحفظ المنشورات الشائعة أو المطلوبة بكثرة، أو ال normal والتي تحفظ المنشورات الاعتيادية...
- في الطبقة الثالثة يتم حفظ البيانات الخاصة بالمستخدم وعلاقاته مع الآخرين
- في الطبقة الرابعة يتم حفظ العمليات التي قام بها المستخدم مثل عمليات الإعجاب أو التعليق ونحو ذلك
- في الطبقة الخامسة والأخيرة يتم حفظ مجاميع الحركات التي قام بها المستخدمون على منشور معين مثل عدد الإعجابات على منشور ما أو عدد الردود ونحو ذلك

والآن نكون وصلنا إلى نهاية رحلتنا في هذا الموضوع، لكن أبق في نفسك أننا تحدثنا عن موضوع واحد من زاوية معينة، لتتمكن من الانطلاق منها وبناء هذا النظام وبما يتوافق مع متطلبات الشركة أو النظام الخاص بك...

فائدة

وإذا كان على المربي أن يتسع صدره لسماع ما يورده عليه الطالب من إشكال واعتراض، فالهوى وحظ النفس قد يقود بعض المربين إلى أن يعد ذلك من سوء الأدب، ومن التطاول على مقام الشيخ والأستاذ، وقد يطلب منه بلسان الحال لا بلسان المقال أن يقبل كل ما يسمع دون سؤال أو مناقشة، وهذا لا يليق بالمربي الحصيف.

ملاحظة مهمة -الفصول اللاحقة-:

ابتداء من هذه الفصول سنقوم بذكر مقتطفات مهمة لتغطية بعض الأفكار أو إغنائك ببعض المصطلحات أو المفاهيم الجديدة، فالفكرة الأساسية قد تم طرحها سابقا، ويمكنك الرجوع للكتاب الأصلي للاستزادة...

DESIGN A CHAT SYSTEM

مقتطفات مهمة:

- أنظمة المحادثة (ال Chat) هي أنظمة قد تكون one-to-one أو Group chat، ومن الأمثلة على ال one-to-one تطبيق ماسينجر، ومن الأمثلة على Group chat تطبيق discord...وطريقة تصميم الشات ليخدم أي فئة من هذه الفئات سيحدد شكل البنية التركيبية للنظام الخاص بك، لكن انتظر، في هذان النوعان لا نعني أن التطبيق لا يحتوي على خاصية التواصل الفردي أو الجماعي، وإنما التركيز أو الهدف الخاص من هذا التطبيق، فماسينجر أو واتس أب يركز على التواصل بين الأشخاص، بينما يركز ال discord على التواصل بين المجموعات، لذلك تجده مشهورا عند مدمني الألعاب...مع أن الواتس اب أو الماسينجر لديهم خاصية إنشاء مجموعات...

- هناك بعض النقاط المهمة عند تصميم برنامج للمحادثة وهي:
  - عدد المستخدمين النشطين
  - عدد الرسائل المرسلة يوميا وطبيعتها (نصية أم متنوعة مثل الفيديو والصور...)
  - معرفة الحالة الخاصة بالمستخدم هل هو online/offline؟ وكيف يمكن إدارة هذه الحالة عند وجود مشاكل في الإتصال بالانترنت
  - كيف سيتم التعامل في حالة وجود أكثر من جهاز متصل لنفس المستخدم، مثلا التطبيق لدي على الهاتف المحمول وعلى اللابتوب (مثال: whatsapp application and whatsapp web)
  - ال Push notification

- قابلية ال system لل scale

- برامج المحادثة يجب أن تهتم بثلاثة أمور رئيسية وهي:
  - القدرة على تلقي الرسائل من قبل الآخرين
  - القدرة على توجيه الرسالة إلى المستقبل الصحيح والقدرة على إيصالها إليه
  - إذا كان المستقبل غير متصل فيجب حفظ الرسائل حتى يصبح متصلا ليتم إرسالها إليه...

- هناك مصطلح مهم وهو ال server-initiated، ويشير هذا المصطلح إلى نموذج اتصال حتى يأخذ ال Server زمام المبادرة لإرسال البيانات لل Client دون أن يطلب ال clients ذلك **صراحة**، أو من خلال خدعة 😛 لنصل إلى هذه البيانات عن طريق ال client، وهناك عدة أساليب تقنية مستخدمة لهذا الغرض نذكر منها:
  - Polling: في هذا الأسلوب يقوم ال client بإرسال request بشكل دوري ومستمر ضمن مدة زمنية محددة إلى ال server حتى يتحقق من عدم وجود أي بيانات جديدة، لكن مشكلة هذا الأسلوب أنه مكلف جدا!، كما أنك دوما ستقوم بعمل ريكوست وليس بالضرورة أن يكون هناك شيء جديد...وهذا إهدار للموارد بلا داعي (خصوصا عندما نتحدث عن كمية ضخمة من ال requests)، ويطلق على هذا الأسلوب: "Regular Polling"، وهناك أنواع أخرى لتقليص المشاكل السابقة مثل: Long Polling و Interval Polling و Intelligent Polling

- ال Websockets: يعد هذا أشهر وأجمل الحلول لمعالجة هذه المشكلة، فال WS وسيلة ممتازة عند حاجتك لإرسال التحديثات بشكل asynchronous من السيرفر إلى ال client، وفكرته ببساطة تقوم على فتح اتصال http من ال client مع السيرفر، ومن ثم جعل هذا الاتصال ثنائي التوجيه، فيمكن لل client إرسال رسالة يستمع لها السيرفر والسيرفر يمكنه إرسال رسالة يستمع إليها ال client، ومن مميزات هذا الأسلوب أنه لا يحتاج إلى أي نوع من التعقيد...فمجرد الاتصال يعني الحصول على اتصال ثنائي التوجيه (bi-directional)

- عند تصميم نظام للشات سيتواجد ثلاثة أجزاء أساسية في التصميم وهي:
  - Stateless Services: مثل الخدمات الخاصة بال login / signup / user profile...إلى آخره
  - Stateful Service: وتستخدم هذه فقط لل Chat service، بحيث نضمن بقاء اتصال ال client مع ال server إلا إن ذهب ال server نفسه! (تحتاج لبقاء الاتصال مستمرا حتى تتمكن من الحصول على التحديثات الجديدة مثل الرسائل أولا بأول)
  - Third-party integration: هناك بعض الخدمات التي ستحتاجها مثل ال Push notification، فلا تنسى تضمينها.

- عادة ما يتم استخدام ال Key / Value store لحفظ الرسائل، وبهذا يمكنك الرجوع للرسائل القديمة أو الحصول على الرسائل الجديدة وحتى وإن لم تكن متصلا بالإنترنت حين إرسالها...

- عادة ما يتم استخدام نوعين من ال Database في مشاريع المحادثة، ال Relational Database وال NoSql database، بحيث يستخدم ال Relational Database لحفظ المعلومات العامة ومعلومات المستخدمين ونحو ذلك، وعند ازدياد أعداد المستخدمين سيكون من الجيد والكافي استخدام ال Replication و sharding التي تحدثنا عنها في الفصل الأول...، أما النوع الثاني سيكون مناسبا لحفظ الرسائل والمحادثات واسترجاعها، واستخدام ال NoSql لهذا الغرض لم يأتي عبثا، بل لحل مشاكل خطيرة ومهمة مثل كمية الرسائل الضخمة والتي لن تحتاج إلى استرجاعها مرة واحدة، أو إمكانية البحث والوصول إلى جزء محدد، كما أن آخر الرسائل غالبا ما يحتاجه المستخدم وتقل أهمية الرسائل بمرور الوقت حتى تنسى -غالبا-!، وهناك مزية جميلة وهي القدرة على ال horizontal scale بشكل سهل كما تحدثنا بالفصل الأول...

- يتم إنشاء ID فريد لكل رسالة، هذا ال ID مهم جدا وقد يشير لمعلومات مهمة مثل أن أكبر أو أجدد ID هو أحدث رسالة ونحو ذلك، وهناك عدة طرق لإنشاء ال ID كما تحدثنا سابقا في الفصل "Design A Unique ID Generator" أو من خلال local sequence number generator، ويتميز هذا الأسلوب بسهولة تنفيذه لأنه سيكون

unique على مستوى group...

- في حالة وجود أكثر من جهاز كيف سنضمن أن كلا الجهازين لديهم آخر نسخة من الرسائل؟ بكل بساطة يتم ذلك من خلال إضافة current_max_id، وهو يشير إلى أكبر أو آخر ID وصل لهذه المحادثة، فإذا كان هناك رقم اللابتوب 50 ورقم الموبايل 100، فنعلم حينها أننا بحاجة لجلب الرسائل لأن الرسائل وصلت إلى الرقم 100 وما زلت أنا عند الرقم 50.

- يتم حفظ حالة المستخدم online/offline في ال Key/Value store، لكن هناك trick جميل، ماذا لو كان المستخدم لديه مشاكل في الاتصال بالانترنت فيتصل ثم يفقد الاتصال ثم يتصل ثم يفقد الاتصال وهكذا بشكل دوري؟!، هذا سيسبب مشكلة وهي نقطة خضراء ثم حمراء ثم خضراء ثم حمراء ^^ وستقول "الشات شلف!"، ويضاف إلى ذلك التكلفة الناجمة عن عملية مثل هذه، ولحل هذه المشكلة يمكن استعارة نظام العمل لدقات القلب "heartbeat"، فدقات القلب تقوم على إصدار نبضة في فترة زمنية محددة، ثم تليها نبضة بعد فترة زمنية وهكذا، فلو قمنا بتطبيق هذا الأسلوب هنا سنعتبر أن إشعار الاتصال هو نبضة، وسنحدد مثلا المدة الزمنية لكل نبضة هي 5 ثواني، فإذا تأخرت نبضات القلب ولم تصلنا لمدة 6 نبضات (30 ثانية) سنقوم بتحويل حالة الاتصال إلى offline ^^

- هناك تفاصيل اضافية يمكنك إضافتها والبحث عنها مثل ال Messages encryption, end to end encryption وال error handling وال cache messaging ^^

فائدة

الرسول الكريم -عليه الصلاة والسلام-؛ بعث معلما ميسرا، فمن رغب في سعادة الدارين، اتبع هدي الحبيب المصطفى -صلى الله عليه وسلم-، والناس في هذا إما مستكثر، وإما مستقل، وإما محروم -والعياذ بالله-!، فاحرص على أن تستكثر العلم بالهدي النبوي ومعرفة سيرته...

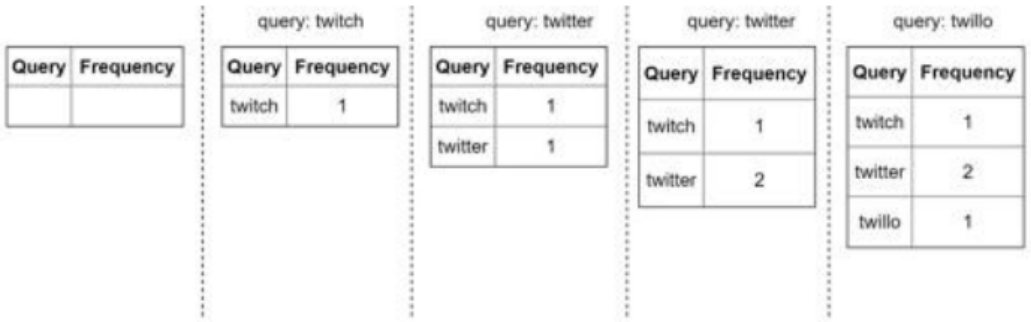
DESIGN A SEARCH AUTOCOMPLETE SYSTEM

مقتطفات مهمة:

- عند بنائك ل AutoComplete system عليك أن تراعي 5 نقاط أساسية وهي:
  - سرعة الحصول على الاقتراحات يجب أن تكون سريعة جدا، وحسب مقال نشره الفيسبوك حول هذه النقطة، فقد أشار إلى أن الاقتراحات إذا كانت تستغرق وقتا أكثر من 100 millisecond فإن هذا سيعطي تجربة سيئة للمستخدم...
  - يجب أن تكون الاقتراحات ذات علاقة مع ما يتم كتابته
  - يجب أن تكون الاقتراحات مرتبة بناء على خوارزمية واضحة تتناسب مع طبيعة النظام مثل (الترتيب حسب كلمات البحث الشائعة أو حسب الكلمات الأكثر بحثا في منطقة جغرافية معينة أو بناء على منطقة زمنية ... إلخ)
  - يجب أن يكون النظام لديه قدرة على التعامل مع كمية بيانات ضخمة
  - يجب أن تعطى ال Availability أولوية عالية في هذا النظام

- تصميم هذا النظام قائمة على فكرة أساسية مكونة من جزئين وهما:
  - Data gathering: وهنا نهتم في طريقة جمع البيانات ليتم استخدامها في الاقتراحات الخاصة بالبحث، وكفكرة أولية يمكنك استخدام جدول في قاعدة

البيانات يقوم بحفظ الكلمات مع عدد مرات البحث:



Query	Frequency

query: twitch

Query	Frequency
twitch	1

query: twitter

Query	Frequency
twitch	1
twitter	1

query: twitter

Query	Frequency
twitch	1
twitter	2

query: twillo

Query	Frequency
twitch	1
twitter	2
twillo	1

Figure 13-2

- Query: وهنا نهتم بطريقة جلب البيانات بما يتناسب مع ما ذكرناه أعلاه (السرعة والفعالية والترتيب وذات علاقة..)، ويمكن كفكرة أولية التفكير بجلب البيانات من قاعدة البيانات من خلال query بسيطة:

```
SELECT * FROM frequency_table
WHERE query Like `prefix%`
ORDER BY frequency DESC
LIMIT 5
```

Figure 13-4

- الشكل الذي ذكرناه في النقطة السابقة يمكن استخدامه لأنظمة صغيرة أو ذات كمية كلمات أو بحث محدود، أما في الأنظمة الكبيرة أو التي قد تتلقى عمليات بحث كثيرة فهذا لن يعمل على الإطلاق، وهنا يأتي دور بعض الأفكار لحل المشاكل، فمثلا جلب البيانات من الشكل السابق من Relational database مع كمية بيانات ضخمة يمثل مصيبة ولن يكون فعالا، بل سيكون منتهكا لأهم نقاط هذا النظام وهي السرعة العالية...، وهذا دون المشاكل الأخرى التي قد تترتب على ذلك والخاصة بال scale ونحوها، وهنا يظهر أحد الحلول الجميلة وهو Trie data structure، ويمثل ال Trie

data structure أحد أشكال ال Tree والتي تبنى على شكل root يمثل ال empty string، كل node تقوم بحفظ حرف معين، وال children لهذه ال node قد يصل عددهم إلى 26 (بافتراض أن البحث باللغة الإنجليزية فقط)، شاهد الصورة أدناه:

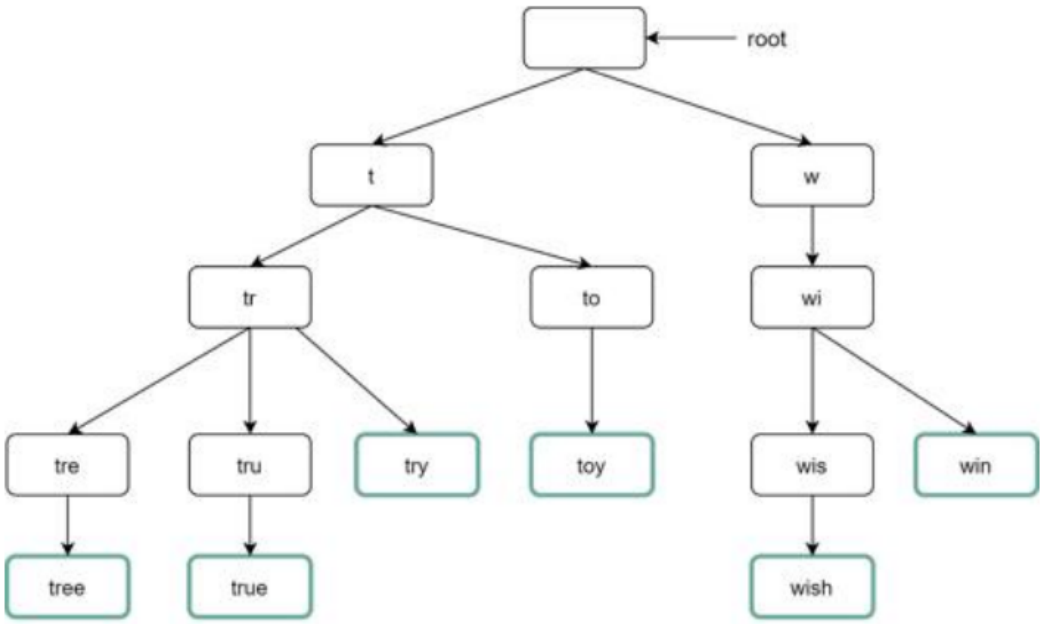


Figure 13-5

ويمكن حفظ عدد التكرارات داخل ال node مثلا true 26 و tree 30 وهكذا، لكن هذا لوحده لا يكفي، لأن عملية جلب البيانات من هذه ال tree بهذا الشكل ستكون بطيئة جدا، لهذا يمكن حل هذه المشكلة من خلال تحديد Limit the max length of a prefix وال Cache top search queries at each node

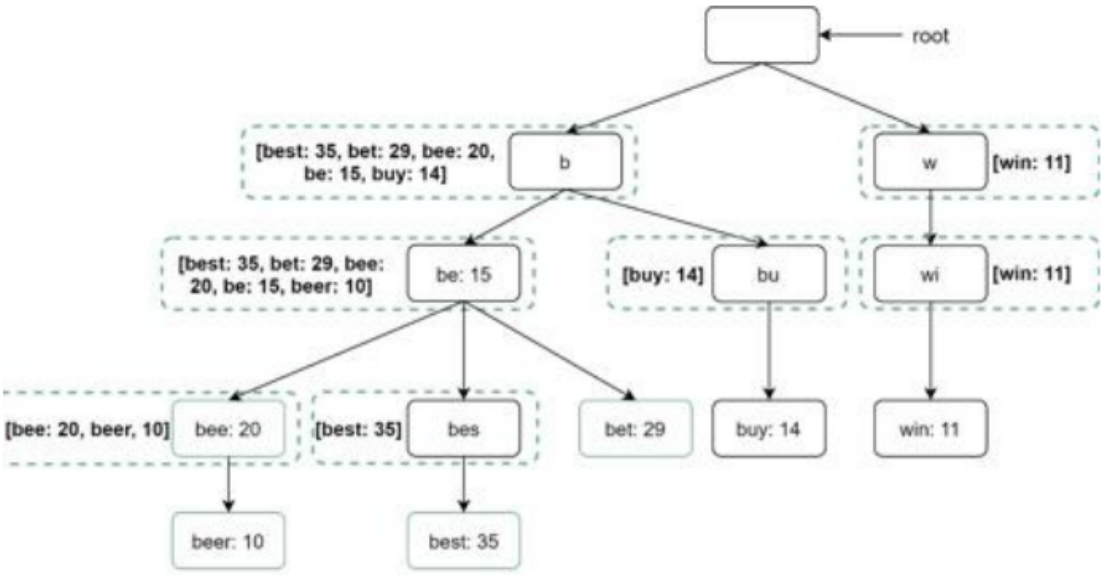


Figure 13-8

- يمكنك الاستفادة من ال Browser Cache لتسريع عملية جلب الاقتراحات بعد المرة الأولى، فالمستخدم الذي قام بالبحث عن كلمة developer قد يقوم بالبحث مجددا عن نفس الموضوع، وهذا تجده مستخدما في محرك البحث جوجل

- يمكن حفظ الاقتراحات لمدة زمنية محددة قبل تحديث الاقتراحات الممكنة، مثلا يمكننا القول كل أسبوع سنقوم بتحديث البيانات، فلو كان هذا الأسبوع Tree أكثر عمليات البحث، فغالبا لن تتغير بشكل كبير، عند الأسبوع التالي ننظر إلى نسخة جديدة من البحث، وهذا يوفر طاقة كبيرة ووقت كبير قد يضيع بلا معنى بعمليات التحديث ودون جدوى حقيقة، ويمكن بناء آلية لضمان أن ال Trend سيتم إضافتها للاقتراحات...

- يمكنك وضع فلتر قبل عملية إرجاع الاقتراحات لمنع الكلمات الفاحشة والبذيئة ونحو ذلك.

- يمكن الاعتماد على ال Unicode لبناء ال nodes الخاصة بال Suggestions tree.

فائدة

الكذب شر غوائل العلم، والتعالم كذب الجاهل أو العالم -المتعالم-، فيكذب الجاهل ليظهر أنه على شيء وهو ليس على شيء، ويكذب العالم ليتعالم على الناس وذلك حتى لا يقال، سئل فلان ولم يعلم!، أو سئل فلان ولم يجب!، وكل ذلك ليحقق منفعة لا يستحقها!، وما نتيجة ذلك التعالم في آخرها إلا فضيحة وخذلان، فعِ ذلك!

DESIGN YOUTUBE

مقتطفات مهمة:

عند تصميمك لنظام يشبه فكرة اليوتيوب عليك مراعاة ما يلي:

- إن تصميم نظام يشبه اليوتيوب ظاهريا بسيط جدا وسهل، فقط تحتاج السماح للمستخدمين برفع الفيديوهات ومشاهدتها!، لكن هذه الجزئية تخفي تعقيدات كبيرة ورائها، ولا يمكن لأي نظام يعمل في فكرة مشابهة لها أن يتجاهلها، ويكفيك أن تقوم بحساب كمية الفيديوهات التي سيتم تحميلها وكمية المساحة المحتاجة لهذا الغرض، وحساب كمية ال streaming التي ستحتاجها والتكلفة المادية المرتبطة بها لتعرف كمية التعقيد المختبئة خلف هذه البساطة، وهذا كله دون التطرق لمواضيع مثل like, comment, devices support, share, download, watch later, view count...إلى آخره

- عند تصميمك لنظام من هذا النوع لديك خيارين اثنين، أن تقوم ببناء cdn & blob storage أو أن تقوم بشراء الخدمة cloud، وهذا القرار مهم جدا وغالبا ستحتاج إلى اتفاقية ما، فتكلفة الاعتماد على cdn بالشكل التقليدي ستكون مكلفة جدا، وعملية إنشاء وبناء هذه الأنظمة من الصفر عملية صعبة ومعقدة وتحتاج إلى تكلفة عالية، وشركات كبيرة مثل فيسبوك تعتمد على cdn وهو Akamai's!

- أهم خصائص هذا النظام تتمحور حول عملية تحميل الفيديو وعملية مشاهدة الفيديو، لذلك، سيتم التركيز على هذه الحيثية في هذه النقطة:

- Upload video flow: هناك عدة أجزاء مهمة في تصميم مخطط سير العمل لتحميل الفيديو، نذكر منها:
  - BLOB Storage: هو مكان ونظام التخزين لحفظ الفيديوهات الأصلية، و BLOB هي اختصار ل Binary Large Object، وفيها يتم حفظ أي بيانات ذات طابع غير بنيوي (unstructured data)، لذلك يمكن في هذا النوع من التعامل مع الصور والفيديوهات وملفات النصوص ونحو ذلك.
  - Metadata DB: يتم حفظ ال metadata الخاصة بالفيديو بها
  - Completion queue & Completion handler: يجب الأخذ بعين الاعتبار أن الفيديوهات التي سيتم رفعها ستدخل إلى Queue، وعند عملية الانتهاء منها سيكون هناك service تمثل ال handler الذي سيأخذ معلومات هذا الفيديو ويحفظها ويعمل تحديث على metadata cache
  - ال Cache وال Load balancer وال API service وال CDN هي أشياء ضرورية ومهمة، لكن لن نتحدث عنها هنا (يكفي ما تحدثنا عنه سابقا، ويمكنك القراءة أكثر حول الموضوع لتغطية بقية الجوانب)
- Streaming video flow: والآن ننطلق إلى الجزء الثاني من مخطط التصميم، فعرض الفيديو للمستخدمين لا يقل أهمية عن طريقة تحميل الفيديوهات، وعملية إدارة البيانات عند العرض لا تقل أهمية عن عملية إدارة أماكن تخزين الملفات ونحو ذلك...ومن هنا ننطلق لما يلي:

- ما يجري عند عرض الفيديو غالبا هي عملية Streaming وليس Downloading!، والفرق بينهما كبير، فال Downloading هي عملية نسخ الفيديو كاملا على الجهاز (Whole Copied)، أما ال Streaming فهي عملية مستمرة ومتواصلة من استقبال البيانات المرسلة من Remote Video إليك، لذلك عند مشاهدتك لفيديو على يوتيوب ترى أن هناك مقدارا صغيرا ومعينا من البيانات تم تحميله، وكلما تقدمت في المشاهدة تم جلب مقدار آخر من البيانات وإضافته للفيديو، بهذا تتمتع بتجربة فريدة من سرعة في المشاهدة وبدون الحاجة لانتظار تحميل الفيديو الكامل!
- عليك أن تعلم أن عملية ال Streaming تتم من خلال استخدام ال Streaming Protocol، وهذه نقطة مهمة، ولدينا عدة Protocols لهذا الغرض وهي MPEG-DASH و APPLE HLS و Adobe HDS و Microsoft Smooth Streaming...، المهم في هذه الجزئية أن تعرف الفرق بين هذه البروتوكولات وما هو البروتوكول المناسب لك، فكل واحد منهم يقدم تشفير مختلف للفيديو وهذا يعني playback مختلف!
- يتم عمل ال Streaming من خلال Video الموجود على CDN مباشرة، ويتم النظر إلى أقرب server لديك يمكنه تقديم هذا الفيديو لتقليل ال Latency قدر الإمكان.

- ال Bitrate هو معدل البت التي تتم معالجتها في زمن معين، وغالبا كلما زاد ال Bitrate زادت دقة الفيديو، واحتاج إلى مساحة أكبر وأضخم للمعالجة وسرعة أكبر للانترنت...
- عليك التأكد من أن الفيديوهات الخاصة بك يمكنها أن تعمل على المتصفحات المختلفة
- للحصول على أفضل تجربة للمستخدم عليك ارسال أعلى جودة من الفيديو يستطيع المستخدم تشغيلها، فالمستخدم صاحب الانترنت السريع والشبكة الجيدة والجهاز الجيد يمكنه أن يشاهد الفيديو بأعلى دقة بشكل افتراضي، بينما المستخدم الذي يملك شبكة سيئة فيتم إرسال جودة منخفضة إليه، وهذه العملية تتم بشكل تلقائي كما تلاحظ في اليوتيوب، خصوصا على الموبايل، فقد تكون في منطقة جيدة ثم تنتقل إلى منطقة ذات اتصال سيء فيتم تحويل جودة الفيديو تلقائيا...
- هناك عدة أنواع ل encoding formats، من أشهرها:
  - Container: وهذا النوع شبيه بالحاوية التي تقوم بتخزين الفيديو والصوت وال metadata الخاصة به، ومن أشهر الأمثلة على هذا mp4 وال mov...
  - Codecs: وهذا النوع يقوم بتشفير وفك تشفير الفيديو بناء على الخوارزمية الخاصة به لتقليل حجم الفيديو مع الحفاظ على دقة الفيديو، ومن أشهر الأمثلة عليه: VP9 و HEVC...

- ما ذكرناه سابقا يندرج تحت باب ال Transcoding a video، لكن هذا لوحدة سيكون مكلفا جدا وتحتاج إلى وقت كبير، لذلك تمت إضافة نموذج آخر يقوم بخدمة احتياجات العملاء، فمثلا هناك أشخاص من أصحاب المحتوى يشترطون وجود watermark مع الفيديو الذي سيتم رفعه...، عمليات مثل هذه بالإضافة لعمليات المعالجة الأخرى التي نحتاجها يمكننا إضافتها من خلال إعطاء القدرة للمبرمج الذي يعمل على مستوى ال client من كتابة وتنفيذ الشيفرة البرمجية المناسبة له، ولهذا الغرض وجد نموذج مثل نموذج ال DAG، وهو النموذج الذي يستخدمه فيسبوك أثناء ال Streaming...أنت هنا تتحدث عن طبقة مجردة ستقوم بالتعامل مع video وكأنه قطعة من عدة أجزاء، لكل جزء عملياته الخاصة...
- هل هذا كل شيء؟!، بكل تأكيد لا، هذه مجرد نقاط عامة عن الموضوع، وهناك الكثير من التفاصيل الدقيقة التي يمكنك قرائتها من الكتاب مباشرة أو من المراجع التي تتحدث عن هذا الموضوع بشكل مفصل، ومن المواضيع التي يجب أن تهتم بها كيف يمكنك تقليل التكلفة قدر الإمكان وكيف يمكنك حماية نفسك وحماية الفيديوهات التي لديك، وكيف يمكنك توزيع السيرفرات لتكون أقرب ما يمكن على المستخدمين، وكيف ستتعامل مع الأخطاء بأنواعها (التي تتم في مرحلة التحميل أو في مرحلة الستريمنج)، ومواضيع ال copyright وكيف ستتعامل معها..إلى آخره

فائدة

لا تستحي من قول لا أدري، فإن لا أدري نصف العلم!، وإن الملائكة -عليهم السلام- "قَالُوا سُبْحَانَكَ لَا عِلْمَ لَنَا إِلَّا مَا عَلَّمْتَنَا ۖ إِنَّكَ أَنتَ الْعَلِيمُ الْحَكِيمُ"، فمن أنا حتى أقول بغير علم؟!، ومن أنا حتى أستحي من أن أقول لا أدري؟!

DESIGN GOOGLE DRIVE

مقتطفات مهمة:

- ال Google Drive واحدة من الخدمات السحابية المنتشرة، والتي تمكنك من مشاركة الملفات بأنواعها وحفظها وإجراء العمليات عليها في مكان واحد، وبغض النظر عن الجهاز المستخدم، الهاتف المحمول أو جهاز الحاسوب!، الجهاز الشخصي أو جهاز العمل...إلخ، وهذا كله يفتح العديد والكثير من الخيارات الممكنة لتصميم مثل هذا النوع من الأنظمة منها: أقصى حجم لملف يمكن رفعه، وعدد المستخدمين النشطين يوميا، وأنواع الملفات التي يمكن رفعها...إلى آخره
- هناك الكثير من النقاط التي ينبغي التركيز عليها مثل طريقة رفع الملفات، وطريقة تحميل الملفات، وكيفية مزامنة الملفات على جميع الأجهزة الأخرى، وإمكانية عرض أو مراجعة سجل التعديلات على الملف، ومشاركة الملفات مع الأصدقاء أو العائلة أو صاحب العمل، وإرسال الإشعارات...
- هناك 5 نقاط رئيسية يجب أن تعتني بها أثناء عملك على هذا المشروع وهي الدقة والمصداقية (Reliability) -فلا مجال لخسارة أي جزء من البيانات-، وسرعة المزامنة، وكمية البيانات المستهلكة (Bandwidth)، وقدرة النظام على التوسع مع وجود كمية مستخدمين أو حركات ضخمة، ويجب أن يكون النظام متاحا دائما وقابلا للعمل حتى لو كانت هناك بعض المشاكل ^^
- عملية رفع الملفات على السيرفر يمكن تقسيمها لجزئين، الأول عملية الرفع التقليدية أو الاعتيادية، والطريقة الثانية هي Resumable upload، ونستخدمها عند وجود حجم ملفات كبير أو / و هناك احتمالية فشل عالية في الشبكة

- التعامل مع مشاكل ال conflict مهم جدا، فاحتمالية وجود تعديل على نفس الملف من أكثر من مستخدم احتمالية واردة، وحدوث تداخل ومشاكل لهذا السبب احتمالية واردة وبقوة، لذلك يمكننا اسنخدام ال sync conflict لحل هذه المشكلة، وبكل بساطة يقوم المبدأ على أن أول تعديل وصل هي عملية ناجحة، والعملية القادمة من المستخدم الثاني ستعود ب sync conflict، ويتم حل هذه المشكلة من خلال إنشاء نسختين من الملف locally للمستخدم 2، واحدة فيها تعديلات المستخدم، والأخرى فيها أحدث نسخة من الملف...وكلما زاد عدد الأشخاص التي تعمل على الملف، زاد التحدي لجعل الملف متزامن...
- هناك مفهوم يطلق عليه Cold Storage، وهو النظام الحاسوبي المسؤول عن تخزين الملفات التي لم يتم زيارتها منذ زمن بعيد...
- أنت بحاجة إلى Offline backup queue بحيث يتم تخزين آخر عمليات المزامنة من الملفات ليتم مشاركتها معك عندما تصبح Online ^^...لكن بالتأكيد ليس كل العمليات، الموضوع يحتاج لتفصيل أكثر...^^
- الملفات التي يتم رفعها على السيرفر تكون على شكل مقاطع صغيرة مشفرة، وهذه العملية تتم داخل ما يسمى ب Block servers، بحيث تقوم هذه السيرفرات بفصل الملفات الكبيرة إلى قطع صغيرة بترتيب واضح...، وهذا الأسلوب يوفر لنا مزية رائعة وهي إمكانية تحميل الجزء المراد تعديله فقط لاحقا واستبداله...وهذا سيوفر الكثير من تدفق البيانات بلا جدوى!

- يجب أن تهتم بموضوع ال Failure، فهنا يمكن حصول الخطأ في أكثر من مكان، لذلك يجب أن تتعامل مع الأخطاء بذكاء بما يتوافق مع الحفاظ على مصداقية البيانات وأكبر قدر من الفعالية والوصول ^^

الخاتمة

وصلنا الآن إلى نهاية رحلتنا في عالم ال System Design في هذا الكتاب، إن تصميم النظم فن وعلم يجمع بين المنطق والإبداع، وتأتي أهميته من قدرته على تحويل الأفكار إلى حقائق ملموسة تلبي احتياجات المستخدمين وتساهم في تطوير البرامج والتطبيقات...

خلال رحلتنا هذه، تعلمنا كيفية تحليل المتطلبات، وتصميم النظام بأسلوب يضمن كفاءة واستدامة المشروع. كما استكشفنا أهمية النمذجة والتخطيط الشامل للأنظمة، وكيفية ضمان الأمان والأداء الموثوق للنظام.

إن تصميم النظم هو تحدي مستمر، حيث تتطور التكنولوجيا وتتغير احتياجات المستخدمين. لذلك، أحثكم على مواصلة التعلم والبقاء على اطلاع بأحدث التطورات في هذا المجال الديناميكي، واعلم أن هذا الكتاب لا يمثل إلا شرح مختصر ومقدمات لهذا العلم، لكنك الآن -بإذن الله تعالى- قادر على أن تبدأ وتبحث في تفاصيل الأمور الدقيقة بما يلبي احتياجاتك!

أخيرًا،

اللهم اغفر لي ولوالدي، ربي ارحمهما كما ربياني صغيرا.

اللهم اغفر لي وللمؤمنين، واغفر اللهم ربنا لإخواننا الذين سبقونا بالإيمان.

اللهم فرج عن إخواننا في فلسطين والشيشان وأفغانستان والسودان والصين والهند والبوسنة والسويد وفرنسا وفي كل مكان تنهك به دماء المسلمين وأعراضهم، وفرج اللهم عن إخواننا المعتقلين في مشارق الأرض ومغربها، ولا حول ولا قوة إلا بالله...

مع خالص التمنيات بالتوفيق والنجاح في كل ما تقومون به.
أخوكم: أنيس حكمت أبوحميد

وآخر دعوانا أن الحمد لله ربّ العالمين