Ars Digitale
Engineering Notes Series

PERFORMANCE ENGINEERING NOTES

Scalability · Latency · Throughput · Observability

Alessandro Fabri

2026 Edition
info@ars-digitale.com

Notes de Performance Engineering

Alessandro Fabri

Ars Digitale

Notes de Performance Engineering

1.3 – Travail d’un performance engineer

Notes de Performance Engineering

Une référence technique structurée pour l’ingénierie de la performance applicative et système

Alessandro Fabri

2026 Edition

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

Copyright

Version: 1.0
Langue : Français

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

2026 Edition

Préface

Ce guide est conçu comme une référence technique structurée pour l'ingénierie de la performance, avec un accent particulier sur la clarté, la précision et l'utilité pratique.

Il couvre le comportement applicatif et système sous charge, le diagnostic, les goulots d'étranglement, les files d'attente, la concurrence, le comportement d'exécution et la performance au niveau des ressources.

Chaque chapitre est pensé à la fois comme une partie d'un parcours cohérent et comme une référence autonome pour l'analyse technique.

À propos de ce livre

Ce livre a été conçu comme une référence pratique et structurée pour l'ingénierie de la performance.

Il vise à combiner :

- précision technique
- clarté conceptuelle
- raisonnement à l'échelle du système
- utilité opérationnelle
- valeur durable comme ouvrage de référence

L'édition EPUB est optimisée pour la lecture numérique et la navigation par chapitres.

Contact: info@ars-digitale.com

Web: https://www.ars-digitale.com

🌍 **Langue :** Français

Notes de Performance Engineering

Cet index fournit la structure complète **française (FR)** du guide **Notes de Performance Engineering**.

Le guide peut être lu de manière séquentielle ou utilisé comme référence technique.

Cette documentation publiée couvre actuellement le **corps théorique principal du guide**.

Structure du guide

1.1 – Fondements

Cette section introduit les concepts fondamentaux nécessaires pour raisonner sur les performances applicatives et des systèmes.

Elle fournit un modèle conceptuel utilisé tout au long du guide.

Elle définit les principes fondamentaux utilisés dans l'ingénierie de la performance pour l'analyse du comportement des systèmes sous charge.

Table des matières

1.1.1 Throughput, latence, concurrence

Définition

Ce sont les trois dimensions principales utilisées pour décrire les performances d'un système.

- **Throughput** : Quantité de travail effectuée par unité de temps ; nombre de requêtes traitées par unité de temps (ex. requêtes par seconde)
- **Latence** : temps nécessaire pour compléter une requête (temps de réponse)
- **Concurrence** : nombre de requêtes en cours de traitement au même moment

Ces concepts sont fondamentaux dans l'ingénierie de la performance et sont utilisés tout au long du guide pour décrire le comportement des systèmes.

Relation

Ces grandeurs ne sont pas indépendantes entre elles.

Pour un système stable :

- augmenter le throughput augmente typiquement la concurrence
- augmenter la concurrence tend à augmenter la latence
- la latence influence directement le nombre de requêtes restant « in flight »

Cette relation est centrale pour comprendre comment les systèmes se comportent sous charge.

Intuition pratique

Un système peut être vu comme une pipeline de traitement :

- **Input** : les requêtes entrent
- **Execution** : elles sont traitées
- **Output** : elles sortent

À tout moment :

- certaines requêtes sont en cours de traitement (concurrence)
- de nouvelles requêtes arrivent (throughput)

- chaque requête nécessite du temps pour être complétée (latence)

Ce modèle mental aide à raisonner sur le flux, l'accumulation et les délais dans les systèmes réels.

Exemple

Si un système traite :

- `100` requêtes par seconde (100 Req./sec.)
- chaque requête nécessite `200 ms` (0.2 s)

alors, en moyenne :

- environ `20` requêtes sont `in flight` à un instant donné

Cette relation est formalisée par la **Loi de Little** :

→ 1.2.1 Loi de Little

Interprétation pratique

Le throughput, la latence et la concurrence forment un système fermé.

Modifier l'un d'eux impacte nécessairement les autres.

Par exemple :

- réduire la latence réduit la concurrence à throughput constant
- augmenter le throughput augmente la concurrence si la latence reste constante
- une forte concurrence augmente la probabilité de mise en file et de contention

Cela constitue un élément clé pour diagnostiquer des problèmes de performance.

1.1.2 Temps de service vs temps de réponse

Définition

Au niveau d'une ressource, le temps de réponse est composé de deux parties :

- **temps de service (S)** : temps consacré à l'exécution effective du travail
- **temps d'attente (Wq)** : temps passé en attente avant d'être traité

Cette distinction est fondamentale dans l'analyse des performances.

Relation

Le temps de réponse (Response Time) :

- inclut à la fois l'`exécution` et l'`attente`
- il augmente lorsque des files d'attente se forment

Même si le temps de service reste constant :

- le temps de réponse peut augmenter significativement en raison de l'attente

C'est l'une des principales raisons pour lesquelles les systèmes se dégradent sous charge.

Signification pratique

Un système lent ne l'est souvent pas parce que le travail est coûteux, mais parce que le travail est en attente de ressources disponibles.

À mesure que la charge augmente :

- les files d'attente s'allongent
- l'attente domine
- le temps de réponse se dégrade

Cette décomposition est formalisée comme suit :

→ 1.2.3 Temps de service vs temps de réponse

Interprétation pratique

Séparer le `temps de service` du `temps de réponse` permet de :

- identifier si le système est limité par le CPU ou par les files d'attente
- distinguer le coût de traitement de la contention sur les ressources
- comprendre si l'optimisation doit cibler l'exécution ou l'attente

Dans de nombreux systèmes réels, les problèmes de latence sont principalement causés par la mise en file plutôt que par le calcul.

1.1.3 Systèmes sous charge

Définition

Un système sous charge traite un flux continu de requêtes entrantes.

La charge est généralement exprimée en :

- requêtes par seconde
- utilisateurs concurrents
- transactions par seconde

La charge définit les conditions opérationnelles dans lesquelles les performances doivent être évaluées.

Comportement

À mesure que la charge augmente :

- l'utilisation des ressources augmente
- des files d'attente commencent à se former
- la latence augmente
- le throughput finit par se stabiliser ou se dégrader

Ces effets ne sont pas linéaires et dépendent de la conception du système et des contraintes des ressources.

Observation clé

Les systèmes ne se dégradent pas de manière linéaire.

À faible charge :

- les performances sont stables

À proximité de la saturation :

- de faibles augmentations de charge peuvent provoquer des augmentations importantes de la latence

Ce comportement non linéaire est une caractéristique clé des systèmes réels.

Interprétation pratique

Comprendre le comportement du système sous charge est essentiel pour :

- le capacity planning
- les tests de performance
- le diagnostic des problèmes de latence

Cela peut expliquer pourquoi les systèmes peuvent sembler stables lors des tests mais échouer avec une charge de production légèrement plus élevée.

1.1.4 Saturation et goulets d'étranglement

Saturation

Une ressource est saturée lorsqu'elle est occupée la plupart du temps ou en permanence.

Exemples typiques :

- CPU à 100 % (ou presque…)
- pool de threads entièrement utilisé
- pool de connexions épuisé

La saturation indique qu'une ressource ne peut pas gérer une demande supplémentaire sans dégradation.

Goulet d'étranglement

Le goulet d'étranglement (bottleneck) est la ressource qui limite le throughput du système.

Caractéristiques :

- utilisation maximale
- files d'attente les plus longues
- contribution dominante au temps de réponse

Le goulet d'étranglement détermine la capacité globale du système.

Signification pratique

Améliorer des ressources qui ne sont pas problématiques (goulets d'étranglement) a peu ou aucun effet.

Les améliorations de performance nécessitent :

- identifier le goulet d'étranglement
- réduire sa demande ou augmenter sa capacité

C'est un principe clé de l'ingénierie de la performance.

Interprétation pratique

Dans les systèmes complexes :

- plusieurs ressources peuvent sembler limitantes
- mais en général une seule limite le throughput à un instant donné

Identifier correctement le goulet d'étranglement est essentiel pour éviter des optimisations inefficaces.

1.1.5 Pourquoi les systèmes ralentissent

Mécanismes courants

La dégradation des performances est généralement induite par un nombre limité de facteurs :

- mise en file due à la saturation
- contention sur des ressources partagées
- utilisation inefficace des ressources
- dépendances externes devenant lentes

Ces mécanismes interagissent souvent et s'amplifient mutuellement.

Effet de la mise en file

Lorsque l'utilisation d'une ressource approche de ses limites :

- le temps d'attente augmente rapidement
- le temps de réponse est dominé par la mise en file

Ce comportement est étroitement lié à l'utilisation et aux effets de file d'attente :

→ 1.2.2 Loi dutilisation

Effets d'amplification

Certains patterns amplifient les problèmes de performance :

- les retries augmentent la charge sur des systèmes déjà saturés
- les timeouts entraînent du travail dupliqué
- les dépendances en cascade propagent les délais

Ces effets peuvent transformer une charge modérée en une dégradation sévère.

Interprétation pratique

La dégradation des performances est rarement causée par un seul facteur.

Elle émerge plutôt de :

- interactions entre composants
- accumulation du temps d'attente
- boucles de rétroaction sous charge

De cela découle la possibilité d'un diagnostic efficace.

Conclusion pratique

La plupart des problèmes de performance ne sont pas causés par une seule opération problématique ou lente, mais par :

- interactions entre composants
- accumulation des temps d'attente
- conditions de surcharge

Comprendre ces mécanismes est essentiel avant d'appliquer des formules ou d'exécuter des tests.

Idée clé

Les performances d'un système sont déterminées par les interactions entre charge de travail, ressources et concurrence.

La compréhension de ces interactions constitue le fondement de l’ingénierie de la performance.

◀ Notes de Performance Engineering | ▲ Index | 1.2 – Métriques et formules de base ▶

1.2 – Métriques et formules de base

Ce document présente une référence synthétique des principales formules utilisées dans la **performance engineering applicative + système**.

Ces formules formalisent les concepts introduits dans :

→ 1.1 Fondements

Elles doivent être lues comme un complément au modèle conceptuel, et non de manière isolée.

Elles fournissent la base quantitative utilisée pour raisonner sur le comportement des systèmes, valider des hypothèses et interpréter les résultats des tests de performance.

Table des matières

Notation (typique)

Symbol	Definition
`X` or `λ`	**throughput** / taux d'arrivée (requêtes par seconde)
`R` or `W`	**temps de réponse** / temps dans le système (secondes)
`S`	**temps de service** sur une ressource (secondes par requête)
`U`	**utilisation** d'une ressource (0–1)
`L`	**concurrence moyenne** / requêtes in flight (comptage)
`V`	**nombre moyen de visites** à une ressource par requête
`D`	**demande de service** sur une ressource (secondes par requête)

Cette notation est utilisée de manière cohérente dans tout le guide et permet d'appliquer les formules de manière uniforme dans des contextes différents.

1.2.1 Loi de Little (concurrence au niveau système)

Définition

Cette loi met en relation la **concurrence** moyenne avec le **throughput** et le **temps dans le système**.

Formule

$L = \lambda \cdot W$

Où

- `L` = nombre moyen de requêtes dans le système (in-flight / concurrence)
- `λ` = taux d'arrivée / throughput (requêtes/s)
- `W` = temps moyen dans le système (s) (souvent le temps moyen de réponse end-to-end)

Signification pratique

Si l'on connaît le `throughput` et le `temps moyen de réponse`, on peut estimer le nombre de requêtes qui sont, simultanément, "in flight" dans le système.

Cela fait de la Loi de Little l'un des outils les plus utiles pour raisonner sur la charge et la concurrence d'un système.

Exemple

Si `λ = 200 req/s` et `W = 0.15 s` :

$L = 200 \cdot 0.15 = 30$

En moyenne, il y a environ **30** requêtes in flight.

Interprétation pratique

La Loi de Little relie trois grandeurs observables :

- throughput
- latence
- concurrence

Cela permet de :

- estimer la concurrence à partir de mesures
- valider le comportement du système
- détecter des incohérences dans les métriques

Cette loi est largement utilisée dans la performance engineering, le capacity planning et le diagnostic des systèmes.

1.2.2 Loi d'utilisation (temps occupé au niveau ressource)

Définition

L'utilisation est la **fraction de temps** pendant laquelle une *ressource unique* est occupée durant un intervalle de temps fixe (typiquement 1 seconde).
Elle mesure le « pourcentage de temps occupé ».

Formule

$U = X \cdot S$

Où

- `U` = utilisation (0–1)
- `X` = throughput observé par cette ressource (req/s)

- `S` = temps moyen de service sur cette ressource (s/req)

Ressource

Une **unité de service unique**, par ex. cœur CPU, thread/worker, connexion DB, etc.

Exemple

Un worker DB traite `50 req/s`, chaque requête nécessite `10 ms = 0.01 s` :

$U = 50 \cdot 0.01 = 0.5 \Rightarrow 50\%$

Interprétation : la ressource est occupée **0.5 seconde par seconde**.

Interprétation pratique

L'utilisation est un indicateur clé de la saturation d'une ressource.

Lorsque l'utilisation s'approche de 1 :

- la mise en file (Queueing) augmente
- la latence croît de manière non linéaire
- la stabilité du système diminue

Cela en fait l'un des signaux les plus importants dans le diagnostic des goulets d'étranglement (bottlenecks).

1.2.3 Temps de service vs temps de réponse (mise en file)

Définition

Le temps de réponse (Response Time) sur une ressource inclut :

- le temps de service (travail effectif)
- le temps de file (attente)

Formule

$R = S + W_q$

Où

- `R` = temps de réponse sur la ressource
- `S` = temps de service
- `W_q` = temps d'attente en file

Signification pratique

Lorsque l'utilisation s'approche de la saturation, la mise en file (Queueing) croît de manière non linéaire et **domine** le temps de réponse, provoquant une **latence long-tail**.

Interprétation pratique

Cette formule explique pourquoi les systèmes ralentissent sous charge même lorsque le coût computationnel ne change pas.

Dans de nombreux systèmes réels :

- le temps de service reste relativement stable
- le temps d'attente augmente rapidement

Par conséquent :

- le temps de réponse est dominé par la mise en file

- la latence devient imprévisible

C'est un point clé dans le diagnostic des problèmes de performance.

1.2.4 Demande de service (visites × temps de service)

Définition

Service total requis d'une ressource par requête, en tenant compte de visites multiples.

Formule

$D = V \cdot S$

Où

- `D` = demande de service sur la ressource (s)
- `V` = visites moyennes à la ressource par requête
- `S` = temps de service par visite (s)

Exemple

Une requête exécute `V = 3` requêtes DB, chacune nécessite `S = 5 ms = 0.005 s` :

$D = 3 \cdot 0.005 = 0.015$ s = 15 ms

Interprétation pratique

La demande de service représente le travail total requis d'une ressource pour chaque requête.

Elle est particulièrement utile pour :

- identifier les ressources les plus utilisées
- estimer les limites de capacité
- comprendre le comportement en montée en charge

Réduire la demande de service est souvent plus efficace qu'augmenter la capacité brute.

1.2.5 Throughput

Définition

Requêtes complétées par unité de temps.

Formule

Formule : `X = N / T`

Où

- `N` = nombre de requêtes complétées
- `T` = fenêtre d'observation (secondes)

Interprétation pratique

Le `throughput` est l'un des principaux indicateurs des performances d'un système.

Il reflète la capacité du système à traiter du travail.

Cependant, le throughput doit toujours être interprété avec :

- la latence

- le taux d'erreur
- l'utilisation des ressources

Un throughput élevé, à lui seul, ne garantit pas un comportement acceptable du système.

1.2.6 Taux d'erreur

Définition

Fraction des requêtes qui échouent (timeouts, 5xx, etc.).

Formule

Formule : `ErrorRate = (N_err / N_total) × 100%`

Interprétation pratique

Le taux d'erreur reflète la fiabilité du système sous charge.

Une augmentation du taux d'erreur indique souvent :

- des conditions de surcharge
- un épuisement des ressources
- de l'instabilité

Le taux d'erreur doit toujours être surveillé avec la latence et le throughput.

1.2.7 Percentiles (p50, p95, p99)

Définition

Le percentile `p`-ième est la valeur en dessous de laquelle se trouve **p% des observations**.

- `p50` ≈ médiane (« requête typique »)
- `p95` = seuil pour les 5% les plus lents
- `p99` = seuil pour les 1% les plus lents

Les percentiles capturent mieux la **distribution** et le **comportement de queue** que les moyennes.

Interprétation pratique

Les percentiles sont essentiels pour comprendre l'expérience réelle de l'utilisateur.

Dans de nombreux systèmes :

- la latence moyenne paraît acceptable
- la latence de queue (p95/p99) est significativement pire

Cette différence est critique pour l'évaluation du système et la définition des SLO.

1.2.7.1 Comment calculer un percentile (échantillon ordonné)

Étant données `N` valeurs triées par ordre croissant :

$v_1 \leq v_2 \leq \ldots \leq v_N$

Calculez la position théorique :

Formule : `P = (p / 100) × (N + 1)`

- Si `P` est un entier → percentile = `v_P`
- Sinon, posez `k = floor(P)` et `δ = P - k` (partie fractionnaire), puis interpolez :

Percentile$(p) \approx v_k + \delta \cdot (v_{k+1} - v_k)$

> Note : les définitions du percentile varient légèrement selon les outils. Cette méthode est une approche couramment utilisée.

1.2.7.2 Interprétation vs moyenne (pourquoi les queues comptent)

- Si `p50` est bien inférieur à la moyenne, la distribution est **asymétrique à droite** (quelques requêtes lentes gonflent la moyenne).
- Si `p95` ou `p99` est très au-dessus de la moyenne, vous avez une **latence long-tail**.

Un pattern typique :

- la moyenne semble « acceptable »
- `p95/p99` sont mauvais

→ l'expérience utilisateur est dégradée pour une fraction non négligeable d'utilisateurs et les SLO sont en risque.

Interprétation pratique

Les percentiles mettent en évidence des comportements que les moyennes masquent.

Ils sont essentiels pour :

- définir les objectifs de niveau de service (SLO)
- détecter les problèmes de latence de queue
- comprendre le comportement dans le pire cas

Ignorer les percentiles conduit souvent à des conclusions incorrectes sur les performances du système.

1.2.8 CDF empirique (seuil → pourcentage)

Définition

Étant donné un seuil `t`, la fonction de distribution cumulative empirique (CDF) indique la fraction d'échantillons inférieurs ou égaux à `t`.

Formule

Formule : `F(t) = count(x_i ≤ t) / N`

Signification pratique

La CDF répond à la question : « Si mon SLO est `200 ms`, quel % de requêtes le respecte ? »

Les percentiles répondent à la question inverse : « Quel seuil correspond à 95% des requêtes ? »

Interprétation pratique

La CDF et les percentiles sont des vues complémentaires des mêmes données.

- CDF : étant donné un seuil → quelle fraction le respecte
- Percentile : étant donnée une fraction → quel seuil lui correspond

Les deux sont utiles pour l'analyse des performances et la validation des SLO.

1.2.9 Latence long-tail (ce que c'est)

Définition

Une petite fraction des requêtes (ex. 5% ou 1%) est **beaucoup plus lente** que la majorité.

Pourquoi la queue « domine »

- Les SLO sont typiquement définis sur `p95/p99`, donc les queues déterminent le pass/fail.
- Dans les systèmes distribués, la dépendance la plus lente détermine souvent la latence end-to-end.
- Les événements de queue sont fréquemment pilotés par la **contention / mise en file**.

Causes communes (haut niveau)

- saturation du thread pool / connection pool (mise en file)
- contention sur les verrous / points chauds de synchronisation
- requêtes DB lentes, index manquants, attentes sur verrou
- retries + timeouts qui amplifient la latence de queue
- hot keys dans les caches / charge non uniforme sur les shards
- pauses GC / pression mémoire (stop-the-world)
- jitter réseau / perte de paquets / retransmissions
- pics de disque I/O, compactions, flush fsync/wal

Interprétation pratique

La latence long-tail est l'un des aspects les plus critiques des performances d'un système.

Elle explique pourquoi :

- les métriques moyennes peuvent paraître acceptables
- l'expérience utilisateur est malgré tout dégradée

Gérer la latence de queue est souvent plus important qu'améliorer la performance moyenne.

1.2.10 Checklist rapide (quoi mesurer dans les tests)

- Latence : `p50/p90/p95/p99`
- Throughput : `RPS/TPS`
- Taux d'erreur : `timeouts/5xx`
- Utilisation : CPU, mémoire, DB, pools
- Longueurs des files : thread pools, connection pools, backlog des messages
- Temps des dépendances : DB/Redis/API externes

Interprétation pratique

Ces métriques constituent l'ensemble minimal requis pour comprendre le comportement du système pendant les tests de performance.

Elles permettent de :

- identifier les goulets d'étranglement
- détecter des instabilités
- corréler la charge de travail avec le comportement du système

Mesurer seulement un sous-ensemble de ces métriques conduit souvent à une analyse incomplète ou trompeuse.

Idée clé

Les formules ne sont pas des abstractions isolées.

Ce sont des outils utilisés pour expliquer le comportement observé et valider les modèles du système.

Leur évaluation constitue un élément essentiel de la performance engineering.

◀ 1.1 – Fondements | ▲ Index | 1.3 – Travail d'un performance engineer ▶

1.3 – Travail d'un performance engineer

Cette section décrit ce qu'est, en pratique, la performance engineering et comment elle est appliquée aux systèmes réels.

Table des matières

1.3.1 Ce qu'est la performance engineering (en pratique)

Définition

La performance engineering est la discipline qui consiste à comprendre, mesurer et contrôler la manière dont un système se comporte sous charge.

Elle ne se limite ni au performance testing, ni à des outils ou technologies spécifiques.

Elle se réfère plutôt à une méthodologie globale de raisonnement sur les systèmes sous charge ou, éventuellement, sous stress.

Elle se concentre sur le comportement global du système et non sur des métriques isolées ou sur des composants individuels.

Performance et exigences non fonctionnelles

La performance engineering ne se concentre pas sur une seule propriété.

Lorsqu'un système est exercé sous charge, un sous-ensemble d'**exigences non fonctionnelles (NFR)** devient visible :

- latence et throughput (performance)
- scalabilité (verticale et horizontale)
- stabilité et résilience sous stress
- utilisation des ressources et efficacité
- limites de capacité

Ces propriétés ne sont pas indépendantes entre elles.

Elles émergent toutes ensemble à mesure que le système est amené à ses limites.

La charge agit comme une **forcing function** qui révèle la manière dont le système se comporte.

Un système qui paraît parfaitement équilibré à faible charge peut montrer un comportement complètement différent lorsqu'il est stressé.

Ce que la performance engineering observe réellement

Sous charge, un système révèle :

- comment le travail traverse ses composants
- comment les ressources sont consommées

- où apparaissent des contentions (contention)
- où se forment les files (queueing)
- quelles sont les limites qui sont atteintes en premier

Cela requiert :

- comprendre le modèle du système (→ 1.1 Fondements)
- mesurer les métriques clés (→ 1.2 Métriques et formules de base)
- identifier les facteurs limitants

L'objectif, de manière évidente, n'est pas seulement d'observer le comportement du système, mais aussi de l'expliquer.

Pas seulement du testing

La performance engineering est souvent réduite au seul **load testing**.

En pratique, la phase de testing n'est qu'une partie du travail.

Les tests sont utilisés pour :

- exposer le comportement du système
- valider des hypothèses
- reproduire des problèmes

Mais la performance engineering comprend également :

- analyser le design du système
- investiguer des problèmes de production
- dimensionner les ressources (heap, pools, threads, connexions)
- expliquer le comportement observé

Le testing sans analyse produit des données sans compréhension.

Perspective pratique

Dans les scénarios réels, le travail implique typiquement :

- préparer et calibrer les environnements de test
- interpréter les exigences non fonctionnelles (NFRs)
- identifier et définir des scénarios de test (significatifs) par rapport aux NFRs
- valider le comportement avec des cas d'usage contrôlés
- appliquer de la charge ou du stress pour faire émerger les problèmes (souvent en white-box)
- identifier et corriger les goulets d'étranglement
- dimensionner les composants du système (CPU, mémoire, pools, limites de concurrence)
- ajuster les configurations et les paramètres (Tuning)
- exécuter des benchmarks pour établir des points de référence
- exécuter des tests de longue durée (soak / endurance) pour valider la stabilité dans le temps

Ces activités ne sont pas isolées.

Elles font partie d'un processus continu orienté vers la compréhension des possibilités et des limites du système.

Idée clé

La performance engineering ne consiste pas (seulement) à rendre un système plus rapide et plus performant.

Elle comprend au contraire un ensemble d'activités et de tâches visant à comprendre comment un système se comporte sous charge de travail, et à s'assurer qu'il reste :

- prévisible
- stable
- scalable

La plupart des problèmes ne sont pas causés par une seule opération « lente », mais par :

- interactions entre composants
- accumulation des temps d'attente
- saturation de ressources partagées

Ces mécanismes constituent, ensemble, le noyau de la performance engineering.

1.3.2 Workflow typique

La performance engineering est un processus itératif dans lequel le système est progressivement exercé, analysé, stabilisé et compris sous des niveaux croissants de charge.

L'objectif n'est pas seulement de détecter des problèmes, mais de construire un modèle fiable de la manière dont le système se comporte dans des conditions réalistes (et limites) de production.

1.3.2.1 Préparation et calibration de l'environnement

- vérifier et aligner l'environnement de test sur les caractéristiques de la production (autant que possible)
- vérifier les configurations (CPU, mémoire, pools, connexions)
- garantir l'observabilité (métriques, logs, traces)

Objectif :

- établir une baseline fiable
- garantir la répétabilité des résultats

Sans calibration, les mesures sont difficiles (ou impossibles) à interpréter et les comparaisons deviennent à tout le moins peu fiables.

1.3.2.2 Définition des cas d'usage et modélisation du workload

Avant d'appliquer de la charge au système, le workload doit être défini.

Un système n'est pas testé en isolation, mais à travers les requêtes qu'il traite.

Cela requiert l'identification précise de :

- les chemins critiques utilisateur et système
- les opérations typiques (read, write, batch, background job)
- la fréquence relative de chaque opération
- les patterns de concurrence

Un workload réaliste inclut :

- un mix de cas d'usage (Use Cases)
- une distribution pondérée (ex. pourcentages de trafic)
- différents types de requêtes et différents coûts

La définition du workload est l'une des étapes les plus critiques et doit être réalisée en étroite collaboration avec ceux qui définissent les exigences non fonctionnelles (NFRs).

Un workload incorrect mène à des conclusions trompeuses, voire totalement inutiles.

Exigences non fonctionnelles (NFR)

En parallèle avec la définition du workload, les **exigences non fonctionnelles** doivent être clarifiées.

Elles définissent ce qui est considéré comme un **comportement acceptable du système**.

Exemples typiques :

- objectifs de throughput (ex. 30 req/s)
- niveaux de concurrence (ex. 500 utilisateurs concurrents)
- objectifs de latence (ex. p95 < 200 ms)
- seuils du taux d'erreur
- contraintes sur l'utilisation des ressources

Les NFR peuvent être :

- définies explicitement par les parties prenantes
- définies seulement partiellement
- manquantes ou incohérentes

Dans tous les cas, elles doivent être :

- réexaminées
- validées
- rendues mesurables

Implication pratique

Le workload et les NFR doivent être alignés.

Pour chaque cas d'usage :

- la charge attendue doit être définie
- le comportement acceptable doit être connu

Sinon :

- les résultats ne peuvent pas être évalués
- les tests ne peuvent être considérés ni comme réussis ni comme échoués

Une définition incorrecte du workload ou l'absence de NFR conduit à des résultats techniquement corrects, mais non actionnables.

1.3.2.3 Tests initiaux de charge / stress (découverte des problèmes)

La première phase de « load test » vise à faire émerger une baseline de référence et d'éventuels problèmes principaux.

Objectifs typiques :

- identifier des goulets d'étranglement évidents
- détecter des erreurs fonctionnelles sous charge
- faire émerger des instabilités (timeouts, crashs, saturation)

Cette phase est souvent :

- exploratoire
- itérative
- partiellement white-box (en utilisant une visibilité interne)

L'objectif est la découverte, non la précision.

1.3.2.4 Analyse et identification des goulets d'étranglement

Une fois que les problèmes ont émergé, le système doit être analysé en détail.

Cela implique :

- corréler les métriques (latence, throughput, utilisation)
- identifier où le temps est dépensé
- localiser les points de saturation et les files

Questions typiques :

- quelle ressource est saturée ?
- où la latence s'accumule-t-elle ?
- qu'est-ce qui limite le throughput ?

Cette étape s'appuie sur :

→ 1.1 Fondements
→ 1.2 Métriques et formules de base

1.3.2.5 Corrections et validation itérative

Après avoir identifié les goulets d'étranglement, les correctifs doivent être appliqués.

Ils peuvent inclure :

- modifications du code
- mises à jour de configuration
- ajustements des ressources (scalabilité verticale/horizontale)

Chaque correction doit être validée en relançant les tests.

Cela crée une boucle itérative :

- **Test → Analyse → Corrige → Teste** à nouveau

L'objectif est de stabiliser progressivement le système.

1.3.2.6 Validation intermédiaire (baseline stable)

Avant de passer à des tests ultérieurs et de longue durée, le système doit atteindre une baseline stable.

Cela signifie :

- aucune erreur critique sous la charge attendue
- comportement prévisible
- latence et taux d'erreur sous contrôle

Cette phase garantit que :

- les problèmes principaux sont résolus
- les résultats sont reproductibles

1.3.2.7 Validation de longue durée (soak / endurance)

Une fois qu'on s'est assuré que le système est stable, il doit être investigué dans la durée.

Cette phase évalue le comportement du système sous une charge de travail soutenue dans le temps.

Objectifs typiques :

- détecter des memory leaks lents
- observer l'accumulation de ressources (threads, connexions, buffers)
- identifier des dégradations de performance dans la durée

- valider la stabilité à long terme

Cette phase est essentielle parce que certains problèmes :

- n'apparaissent pas immédiatement
- émergent seulement après un exercice prolongé

Les résultats de cette phase ont un impact direct sur :

- dimensionnement du système
- capacity planning
- configuration de runtime

1.3.2.8 Dimensionnement et définition de la capacité

Sur la base des observations précédentes, et aussi à partir d'éventuels tests unitaires successifs à la phase de stabilisation de la baseline, les composants du système sont dimensionnés.

Cette phase inclut :

- configuration du heap et des mémoires
- thread pools et connection pools
- limites de concurrence
- dimensionnement de l'infrastructure
- clustering

L'objectif est de définir :

- quelle charge le système peut supporter
- dans quelles conditions il reste stable
- quelles marges éventuelles sont requises

Le dimensionnement doit se baser sur le comportement observé, et non sur des hypothèses.

1.3.2.9 Tuning

Une fois le dimensionnement défini, le tuning affine le comportement du système.

Domaines typiques :

- paramètres du garbage collector
- scheduling des threads et dimensionnement des pools
- paramètres de la base de données et des connexions
- stratégies de caching

Le tuning vise à :

- réduire la latence
- améliorer la stabilité
- optimiser l'utilisation des ressources

Il est souvent itératif et dépendant du contexte spécifique.

1.3.2.10 Vérification et régression

Après la phase de tuning, le système doit être validé à nouveau.

Cela inclut :

- rejouer les scénarios clés
- vérifier que les améliorations sont effectives
- s'assurer qu'aucune régression n'est introduite

Cette phase garantit cohérence et fiabilité.

1.3.2.11 Benchmarking et points de référence

Enfin, les benchmarks sont établis.

Ils fournissent :

- des métriques de performance de référence
- des points de comparaison entre versions
- une validation par rapport aux attentes

Les benchmarks ne sont pas des objectifs en eux-mêmes.

Ils sont utilisés pour :

- comprendre le comportement du système
- suivre son évolution dans le temps

Idée clé

La performance engineering se développe selon une boucle itérative :

- **définis le workload → teste → analyse → corrige → valide → optimise**

L'objectif n'est pas seulement d'améliorer les performances, mais de comprendre les limites du système et de garantir un comportement prévisible sous charge.

1.3.3 Black-box vs white-box

La performance engineering peut être abordée selon deux perspectives complémentaires :

- **black-box** (observation externe)
- **white-box** (observation interne)

Les deux sont nécessaires pour comprendre le comportement du système sous charge de travail.

1.3.3.1 Approche black-box

Dans une approche black-box, le système est observé depuis l'extérieur.

Seul le comportement visible extérieurement est mesuré :

- temps de réponse
- throughput
- taux d'erreur

L'implémentation interne n'est pas prise en considération.

Ce que cela fournit

L'observation black-box permet de :

- valider le comportement du système du point de vue de l'utilisateur
- mesurer les performances end-to-end
- détecter des erreurs visibles sous charge

Elle répond à des questions telles que :

- Le système est-il suffisamment rapide ?

- Gère-t-il la charge attendue ?
- Échoue-t-il sous stress ?

Limites

Le seul black-box ne peut pas expliquer :

- où éventuellement du temps est le plus souvent dépensé
- quelle ressource est saturée
- pourquoi les performances se dégradent

Il montre les symptômes, non les causes.

1.3.3.2 Approche white-box

Dans une approche white-box, le comportement interne du système est observé.

Cela inclut :

- utilisation des ressources (CPU, mémoire, disque, réseau)
- thread pools et connection pools
- files internes
- temps au niveau des composants

L'observation white-box fournit un niveau d'**introspection dans l'exécution du système**.

Dans de nombreux cas, cela inclut une visibilité proche du niveau du code :

- temps au niveau des méthodes
- call paths et flux d'exécution
- hotspots (méthodes lentes ou exécutées fréquemment)
- patterns d'allocation et comportement mémoire
- contention sur les verrous et points de synchronisation

Ce que cela fournit

L'observation white-box permet de :

- identifier les goulets d'étranglement
- comprendre où le temps est dépensé
- détecter la contention (contentio) et la mise en file (queueing)
- analyser la saturation des ressources

Elle répond à des questions telles que :

- Quel composant est lent ?
- Où la latence s'accumule-t-elle ?
- Qu'est-ce qui limite le throughput ?
- Quelle partie de l'exécution est responsable du ralentissement ?

Limites

Le seul white-box ne garantit pas :

- un comportement end-to-end correct
- une expérience utilisateur acceptable

Un système peut paraître intérieurement efficace, tout en échouant sous des conditions de workload réel.

1.3.3.3 Observabilité et instrumentation

L'observabilité fournit les données nécessaires à l'analyse white-box.

Elle inclut typiquement :

- métriques système et applicatives (ex. utilisation CPU, latence, throughput)
- logs (événements, erreurs, changements d'état)
- traces (flux des requêtes entre composants)
- application performance monitoring (APM)

Ces sources fournissent une visibilité continue sur le comportement du système.

Artifacts diagnostiques

En plus de l'observabilité continue, une analyse plus profonde se base souvent sur des artifacts diagnostiques.

Ceux-ci sont typiquement collectés on demand et fournissent un snapshot de l'état du système.

Exemples courants :

- thread dumps (états des threads, verrous, contention)
- heap dumps (utilisation mémoire, rétention des objets, leaks)
- snapshots de profiling (profiling CPU et allocations)
- core dumps (analyse des défaillances au niveau du processus)

Ces artifacts permettent de :

- inspecter l'état interne de l'exécution
- identifier des threads bloqués et des deadlocks
- analyser des memory leaks et des retention paths
- investiguer en détail des anomalies de performance

Ils sont en général plus lourds et plus intrusifs que les outils d'observabilité, et sont utilisés de manière sélective pendant le diagnostic.

1.3.3.4 Combiner les deux approches

Une performance engineering efficace requiert la combinaison des deux perspectives.

Workflow typique :

- utiliser le black-box pour détecter les problèmes
- utiliser le white-box pour les expliquer
- valider à nouveau les améliorations avec le black-box

Cela crée une boucle de feedback :

- **observe → analyse → corrige → valide**

Idée clé

L'observation **black-box** révèle qu'un problème existe.

L'observation **white-box** explique pourquoi il existe.

Les deux sont nécessaires pour comprendre et contrôler le comportement du système sous charge.

1.3.4 Load testing vs diagnostic

Le load testing et le diagnostic sont souvent confondus.

Ils servent des objectifs différents et opèrent à des niveaux différents.

Les deux sont nécessaires pour comprendre le comportement du système sous charge de travail.

1.3.4.1 Load testing

Le load testing applique un workload contrôlé au système.

Il est utilisé pour :

- observer le comportement dans des conditions spécifiques
- mesurer la latence, le throughput et les taux d'erreur
- valider des hypothèses sur la capacité et la scalabilité

Le load testing opère principalement au niveau **black-box** :

- les requêtes sont générées extérieurement
- les réponses sont mesurées extérieurement

Ce que cela fournit

Le load testing répond à des questions telles que :

- Le système peut-il supporter la charge attendue ?
- Que se passe-t-il lorsque la charge augmente ?
- Quand les performances se dégradent-elles ?
- Quel est le throughput maximal soutenable ?

Limites

Le seul load testing n'explique pas :

- pourquoi le système ralentit
- quel composant est responsable
- comment les ressources sont utilisées en interne

Il révèle le comportement, mais non les causes.

1.3.4.2 Diagnostic

Le diagnostic investigue le comportement interne du système.

Il est utilisé pour :

- identifier les goulets d'étranglement
- comprendre les chemins d'exécution
- analyser l'utilisation des ressources
- expliquer les problèmes de performance observés

Le diagnostic opère au niveau **white-box** :

- des métriques internes sont analysées
- des traces et des chemins d'exécution sont inspectés
- des artifacts diagnostiques peuvent être collectés

Ce que cela fournit

Le diagnostic répond à des questions telles que :

- Où le temps est-il dépensé ?
- Quelle ressource est saturée ?
- Quel composant est responsable de la latence ?
- Qu'est-ce qui cause la dégradation des performances ?

Outils et techniques

Le diagnostic se base typiquement sur :

- métriques, logs et traces
- application performance monitoring (APM)
- thread dumps et heap dumps
- profiling et analyse de l'exécution

Limites

Le diagnostic sans load testing peut ne pas saisir :

- des conditions de workload réel
- des interactions entre composants
- le comportement sous stress

Il peut expliquer un problème, mais pas nécessairement le reproduire.

1.3.4.3 Relation entre load testing et diagnostic

Le load testing et le diagnostic doivent être combinés.

Workflow typique :

- appliquer de la charge pour exposer le comportement
- utiliser le diagnostic pour analyser l'état interne
- appliquer des corrections
- valider à nouveau avec le load testing

Cela crée une boucle :

- observe → explique → corrige → valide

Idée clé

Le load testing révèle qu'un problème existe.

Le diagnostic explique pourquoi il existe.

Aucun des deux n'est suffisant à lui seul.

La compréhension du comportement du système requiert les deux.

1.3.5 Ce qui compte vraiment (et ce qui ne compte pas)

La performance engineering implique un ensemble étendu d'outils, de métriques et de techniques.

Cependant, toutes ne peuvent pas avoir le même niveau d'importance dans des contextes hétérogènes.

Comprendre ce qui compte est essentiel pour éviter de gaspiller des efforts et de tirer des conclusions erronées.

Ce qui compte

Les aspects les plus importants sont :

- **comprendre le comportement du système sous charge**
- **identifier les goulets d'étranglement et les facteurs limitants**
- **utiliser des workloads réalistes et des NFR validés**
- **raisonner sur les interactions entre composants**
- **mesurer et interpréter correctement les résultats**

La performance engineering concerne principalement :

- construire un modèle mental du système
- valider ce modèle à travers des observations
- l'affiner par itération

Ce qui ne compte pas (autant qu'il semble)

Certains aspects sont souvent excessivement mis en avant :

- outils et frameworks
- métriques isolées sans contexte
- scénarios de test synthétiques ou irréalistes
- micro-optimisations sans impact au niveau système
- résultats d'un seul test pris isolément

Ces éléments peuvent être utiles, mais ils ne sont pas suffisants.

Malentendus courants

Divers malentendus apparaissent fréquemment :

- « Si j'exécute un load test, je comprends le système »
- « Si le CPU est bas, le système est sain »
- « Si la latence moyenne est acceptable, le système va bien »
- « Plus de matériel résoudra le problème »

Ces hypothèses conduisent souvent à des conclusions incorrectes.

Pensée au niveau système

Les performances émergent des interactions :

- entre composants
- entre workload et ressources
- entre concurrence et mise en file

Se concentrer sur une seule partie du système est rarement suffisant.

Ce qui est nécessaire, c'est une vision globale.

Implication pratique

Une performance engineering efficace requiert :

- poser les bonnes questions

- valider les hypothèses
- corréler des signaux multiples
- itérer sur la base des preuves

Outils, tests et métriques soutiennent ce processus, mais ne le remplacent pas.

Idée clé

La performance engineering ne consiste pas à collecter des données.

Elle consiste à comprendre ce que les données signifient.

L'objectif n'est pas de produire des chiffres, mais d'expliquer le comportement du système et de prendre des décisions éclairées.

◀ 1.2 – Métriques et formules de base | ▲ Index | 01-04-types-of-performance-tests ▶

1.4 – Types de tests de performance

Ce chapitre introduit les principales catégories de tests de performance utilisées dans la performance engineering.

Chaque type de test de performance répond à une question différente sur le comportement du système sous charge.

Dans leur ensemble, ils aident à évaluer les performances, la stabilité, la scalabilité, la récupération et la capacité du système de manière contrôlée et mesurable.

Table des matières

1.4.1 Objectif du performance testing

Définition

Le performance testing, comme déjà abordé dans les paragraphes précédents, évalue comment un système se comporte dans des conditions de workload contrôlé.

Il fournit des données mesurables sur :

- latence
- throughput
- taux d'erreur
- utilisation des ressources

(→ 1.2 Métriques et formules de base)

Le performance testing n'est donc pas seulement une activité de mesure, mais aussi une activité de validation.

Il est exploité pour comparer le comportement attendu (défini dans les NFRs) avec le comportement observé dans des conditions de workload définies.

Rôle dans la performance engineering

Le performance testing ne concerne pas seulement la mesure des résultats.

Il est utilisé pour :

- valider le comportement du système dans les conditions attendues
- faire émerger des goulets d'étranglement et des limitations
- soutenir le capacity planning
- valider des décisions architecturales

Il fournit également un framework contrôlé pour comparer :

- des versions du même système
- différentes configurations
- des changements d'infrastructure
- des choix de tuning

Sans tests contrôlés, les discussions sur les performances restent souvent fondées sur des hypothèses plutôt que sur des preuves.

Le workload comme modèle

Un workload de test représente un modèle simplifié de l'utilisation réelle.

Il définit :

- taux d'arrivée (requêtes par seconde)
- concurrence (nombre d'utilisateurs ou de requêtes actives)
- patterns des requêtes (distribution, mix d'opérations)

(→ 1.2.1 Loi de Little (concurrence au niveau système))

Un workload n'est pas le miroir exact de l'utilisation réelle de production en soi.

C'est une approximation pratique des patterns d'utilisation les plus pertinents.

Pour cette raison, la valeur d'un test de performance dépend fortement du degré de réalisme du modèle de workload.

Conditions contrôlées

Un test de performance n'est significatif que si les conditions d'exécution sont bien définies et contrôlées.

Cela inclut :

- la définition du workload
- la durée du test
- l'environnement dans lequel il est exécuté
- les métriques collectées pendant l'exécution

Si ces conditions ne sont pas claires, les résultats, bien que toujours numériques, seront peu ou totalement dépourvus de valeur de connaissance et de projection.

Le contrôle des conditions initiales est l'un de ces paramètres qui transforme un test d'un simple exercice en une activité d'ingénierie indispensable.

Le performance testing est donc le point d'entrée de nombreux concepts développés dans la suite de ce document.

Comme pratique globale de test, il fait émerger :

- effets de mise en file et de saturation (→ 1.5 Comportement du système sous charge)
- limites de concurrence (→ 1.6 Concurrence et parallélisme)
- effets de runtime et de mémoire (→ 1.7 Runtime et modèle de mémoire)
- saturation des ressources (→ 1.8 Performances au niveau ressource)

Pour cette raison, le design des tests devrait toujours être relié à une connaissance approfondie et globale du système.

Signification pratique

Un bon test de performance ne répond pas seulement à :

- « À quelle vitesse le système fonctionne-t-il ? »

Il aide aussi à répondre à :

- « Dans quelles conditions le système reste-t-il stable ? »
- « Qu'est-ce qui change à mesure que la charge augmente ? »
- « Quelle limite est atteinte en premier ? »
- « Quel type de dégradation apparaît ? »

Ces questions sont essentielles en performance engineering parce qu'elles relient la mesure à l'interprétation.

Idée clé

Les tests de performance sont des expériences contrôlées.

Ils sont conçus pour observer le comportement du système dans des conditions spécifiques de workload.

Leur valeur ne réside pas seulement dans les mesures qu'ils produisent, mais surtout dans la compréhension qu'ils fournissent.

1.4.2 Load testing

Définition

Le **load testing** évalue le comportement du système sous workload standard ou typique.

C'est la manière la plus commune et la plus directe de valider qu'un système se comporte de manière acceptable dans des conditions opérationnelles normales.

Objectif

- vérifier que le système satisfait les exigences de performance
- valider des objectifs de latence et de throughput
- observer l'utilisation des ressources dans des conditions normales

Le load testing répond à la question de savoir si le système se comporte correctement dans l'intervalle opérationnel qu'il est censé supporter.

Caractéristiques

- le workload est stable et contrôlé
- le système opère dans son intervalle attendu
- l'attention est portée sur le comportement en régime stationnaire

L'objectif n'est pas de pousser le système à ses limites, mais d'établir s'il se comporte correctement sous une charge (de production) pour laquelle il a été conçu.

Exemple

Un système conçu pour :

- 200 requêtes par seconde
- latence p95 < 300 ms

Un load test vérifie que ces objectifs sont satisfaits.

Il peut aussi vérifier que :

- le taux d'erreur reste faible
- le throughput reste stable
- l'utilisation des ressources reste dans des limites acceptables

---

Valeur diagnostique

Le load testing fournit une baseline :

- distribution normale de la latence
- utilisation typique des ressources
- throughput attendu

Cette baseline est essentielle pour la comparaison avec les autres tests.

Sans une baseline fiable, il est difficile de déterminer si le comportement observé dans les tests de stress, spike, soak ou capacity est anormal ou simplement normal pour le système analysé.

Limites du load testing

Le seul load testing ne détermine pas :

- la capacité maximale du système
- les points de rupture du système
- la stabilité à long terme du runtime
- le comportement de récupération après des changements brusques de charge

Un système peut réussir un load test et néanmoins échouer sous surcharge, exécution prolongée ou bursts rapides de trafic.

Pour cette raison, le load testing est nécessaire mais non suffisant.

Interprétation pratique

Le load testing est le point de référence pour le reste de l'analyse de performance.

Il définit le comportement opérationnel normal du système et permet d'interpréter les tests suivants dans leur contexte.

Si le système se comporte déjà mal sous la charge standard, il y a peu de valeur à passer immédiatement à des types de tests plus avancés.

Idée clé

Le load testing répond à : *« Le système se comporte-t-il correctement sous la charge attendue ? »*

Il établit la baseline par rapport à laquelle tous les autres tests de performance peuvent être interprétés.

1.4.3 Stress testing

Définition

Le **stress testing** évalue le comportement du système au-delà de sa capacité attendue.

Il est utilisé pour observer ce qui se passe lorsque le système est poussé hors de son intervalle opérationnel prévu.

Objectif

- identifier les limites du système
- observer le comportement en surcharge
- détecter les modes de défaillance

Le stress testing concerne principalement le comportement à la limite du système et la dégradation des capacités de travail sous une charge excédant les standards prévus.

Caractéristiques

- le workload augmente au-delà des niveaux normaux
- le système s'approche ou atteint la saturation

(→ 1.8 Performances au niveau ressource)

La surcharge peut être appliquée progressivement ou maintenue à un niveau clairement excessif.

Dans les deux cas, l'objectif est de faire émerger la manière dont le système se comporte lorsque la demande dépasse la capacité.

Effets observables

- la latence augmente rapidement
- le throughput se stabilise ou diminue
- le taux d'erreur augmente

(→ 1.5.3 Dégradation non linéaire)
(→ 1.5.4 Effondrement du throughput)

Des effets supplémentaires peuvent inclure :

- accumulation des files
- amplification des timeouts
- épuisement des pools
- utilisation instable des ressources
- surcharge pilotée par les retries

Valeur diagnostique

Le stress testing révèle :

- des goulets d'étranglement
- des points de saturation
- la stabilité du système sous pression

Il est particulièrement utile pour comprendre si la dégradation est graduelle, brutale, récupérable ou instable.

Deux systèmes avec des résultats similaires dans les load tests peuvent se comporter de manière très différente sous stress.

Comportement en rupture

Un aspect important du stress testing n'est pas seulement si et quand le système échoue, mais comment il échoue.

Les questions pertinentes incluent :

- La latence augmente-t-elle avant que des erreurs n'apparaissent ?
- Les erreurs apparaissent-elles progressivement ou soudainement ?
- Le throughput se stabilise-t-il avant de s'effondrer ?
- Le système récupère-t-il lorsque la charge est réduite ?

Ces questions comptent d'un point de vue opérationnel, parce que la surcharge est un scénario réaliste dans les systèmes de production.

Distinction avec le capacity testing

Le stress testing et le capacity testing sont liés, mais différents.

- le **stress testing** se concentre sur le comportement en surcharge et sur les modes de défaillance
- le **capacity testing** se concentre sur la charge maximale soutenable qui satisfait encore les exigences

Le stress testing continue donc au-delà de l'intervalle opérationnel acceptable pour examiner la dégradation et la rupture.

Interprétation pratique

Le stress testing est utile lorsque la question d'ingénierie n'est pas seulement :

- « Quelle charge le système peut-il supporter ? »

mais aussi :

- « Que se passe-t-il après qu'il ne peut plus supporter la charge ? »
- « Se dégrade-t-il de manière progressive ? »
- « Peut-il récupérer proprement ? »

Ce sont des questions essentielles pour la résilience et la robustesse opérationnelle.

Idée clé

Le stress testing répond à : *« Que se passe-t-il lorsque le système est poussé au-delà de ses limites ? »*

Il révèle comment le système se dégrade, comment il échoue et quelle surcharge il peut tolérer avant de devenir instable.

1.4.4 Spike testing

Définition

Le **spike testing** évalue le comportement du système sous des augmentations soudaines de charge.

Contrairement au load testing ou au stress testing graduel, le spike testing se concentre sur les transitions rapides plutôt que sur des conditions opérationnelles stables.

Objectif

- observer la réaction à des changements brusques du workload

- évaluer élasticité et récupération
- détecter une instabilité transitoire

Le spike testing est particulièrement pertinent pour des systèmes exposés à un trafic bursty, à des pics liés à des campagnes, à une demande pilotée par des événements ou à de brèves montées d'activité.

Caractéristiques

- le workload augmente rapidement et en très peu de temps
- le système doit s'adapter rapidement

La caractéristique distinctive n'est pas seulement le volume de charge, mais la vitesse à laquelle la charge change.

Un système peut gérer une charge élevée lorsqu'elle est atteinte progressivement, mais se comporter mal lorsque cette même charge arrive soudainement.

Effets observables

- pics temporaires de latence
- accumulation de files
- erreurs potentielles pendant la transition

(→ 1.5 Comportement du système sous charge)

Des effets supplémentaires peuvent inclure :

- réponse retardée du scaling
- épuisement transitoire des connexions
- cascades temporaires de timeouts
- récupération lente après le burst

Valeur diagnostique

Le spike testing révèle :

- sensibilité au trafic bursty
- comportement de mise en file sous charge soudaine
- capacité de récupération après le spike

Ce type de testing est précieux parce que de nombreux systèmes sont optimisés pour des conditions de régime stationnaire mais restent fragiles pendant des transitions brusques.

Comportement de récupération

La partie la plus importante du spike testing est souvent ce qui se passe après le spike.

Les questions pertinentes incluent :

- Le système revient-il rapidement à la latence normale ?
- Les files se vident-elles de manière contrôlée ?
- Les ressources sont-elles libérées correctement ?
- Le système reste-t-il dégradé après que le spike est passé ?

Un système qui survit au spike mais récupère lentement peut malgré tout être opérationnellement faible.

Interprétation pratique

Le spike testing est particulièrement utile pour des systèmes qui sont :

- exposés extérieurement à un trafic bursty
- dépendants de l'auto-scaling ou d'un comportement élastique
- sensibles à l'accumulation de files
- soumis à des changements de demande pilotés par des événements

Dans ces cas, la charge moyenne est souvent moins importante que les pics de courte durée et que la réaction du système à ceux-ci.

Idée clé

Le spike testing répond à : *« Comment le système réagit-il à des changements soudains de charge ? »*

Il évalue non seulement la résistance aux bursts, mais aussi la capacité à récupérer proprement après ceux-ci.

1.4.5 Soak testing

Définition

Le **soak testing** évalue le comportement du système sur une période étendue sous charge soutenue.

Il est parfois aussi appelé endurance testing.

Son objectif est de faire émerger des problèmes qui n'apparaissent pas dans des tests de courte durée.

Objectif

- détecter des problèmes de long terme
- observer la stabilité dans le temps
- identifier une dégradation graduelle

Le soak testing concerne moins la performance de pointe et davantage la cohérence, l'accumulation et la dérive.

Caractéristiques

- le workload est constant ou varie lentement
- la durée du test est longue (heures ou jours)

La dimension clé est le temps.

Certains systèmes se comportent correctement pendant quelques minutes mais se dégradent après des heures en raison d'effets d'accumulation.

Effets observables

- croissance de la mémoire
- fuites de ressources
- dégradation des performances dans le temps

(→ 1.7 Runtime et modèle de mémoire)

Des symptômes supplémentaires de longue durée peuvent inclure :

- accumulation de threads
- leakage de connexions
- files en lente augmentation
- croissance de l'overhead du GC

- déséquilibre du cache ou rétention incontrôlée

Valeur diagnostique

Le soak testing révèle :

- slow memory leak
- épuisement des ressources
- instabilité de long terme

C'est souvent le seul moyen fiable de valider si le système reste sain et exploitable pendant une activité prolongée.

Cela est essentiel pour les systèmes de production qui doivent fonctionner en continu.

Dégradation dépendante du temps

Le soak testing est important parce que certaines ruptures ne sont pas basées sur des seuils, mais sur le temps.

Des exemples incluent :

- mémoire retenue lentement dans le temps
- pools non complètement libérés
- tasks en background qui accumulent de la dérive
- patterns de retry qui augmentent lentement la pression
- caches qui croissent sans eviction efficace

Ces problèmes peuvent ne pas apparaître dans des load tests ou des stress tests de courte durée.

Valeur opérationnelle

Un système qui se comporte bien pendant dix minutes mais se dégrade après six heures n'est pas stable.

Le soak testing contribue donc directement à :

- validation pour la mise en production
- confiance dans le runtime
- évaluation de la fiabilité de long terme
- dimensionnement de l'infrastructure et du runtime

Il aide aussi à valider que le monitoring reste significatif sur de longues périodes d'opérativité.

Interprétation pratique

Le soak testing est particulièrement important pour des systèmes avec :

- longs uptimes
- traitement en background
- runtimes avec gestion de la mémoire
- architectures riches en connexions
- pools de ressources qui changent lentement dans le temps

Dans de tels systèmes, les résultats de performance de courte durée ne sont pas suffisants pour garantir la stabilité réelle.

Idée clé

Le soak testing répond à : *« Le système reste-t-il stable dans le temps ? »*

Il valide le comportement de longue durée et révèle des problèmes causés par l'accumulation, la dérive et la dégradation lente.

1.4.6 Capacity testing

Définition

Le **capacity testing** détermine le workload maximal qu'un système peut gérer tout en satisfaisant les exigences de performance.

Il est utilisé pour identifier la limite opérationnelle pratique du système dans des conditions acceptables.

Objectif

- identifier le throughput maximal soutenable
- déterminer des limites opérationnelles sûres
- soutenir le capacity planning

Le capacity testing est donc directement relié à la planification, au dimensionnement, au forecasting et aux décisions opérationnelles.

Méthode

- éventuels tests unitaires pour baseline dimensionnelle
- augmenter graduellement le workload
- surveiller latence, throughput et erreurs
- identifier le point où les performances se dégradent

L'augmentation de la charge devrait être contrôlée et mesurable.

Cela permet de localiser la limite du système avec une plus grande précision que dans un stress test purement exploratoire.

Interprétation

La limite de capacité est atteinte lorsque :

- la latence dépasse des seuils acceptables
- le taux d'erreur augmente
- le throughput ne scale plus

(→ 1.2 Métriques et formules de base)
(→ 1.5 Comportement du système sous charge)

En pratique, la limite n'est pas toujours une seule valeur exacte.

Elle peut être mieux comprise comme un intervalle dans lequel le comportement acceptable commence à se détériorer.

Ce que révèle le capacity testing

Le capacity testing révèle :

- la charge soutenable la plus élevée sous des critères d'acceptation définis
- la marge entre la charge attendue et la charge maximale acceptable
- la relation entre demande croissante et comportement dégradé
- le point où une charge supplémentaire ne produit plus de throughput utile

Ces informations sont essentielles pour des décisions d'ingénierie et de planification.

Relation avec le capacity planning

Le capacity testing est l'un des principaux inputs du capacity planning.

Il aide à répondre à des questions telles que :

- Quel trafic le système actuel peut-il supporter ?
- Quelle headroom est disponible ?
- Quand faudra-t-il scaler ?
- Quel composant contraint en premier la capacité ?

Cela rend le capacity testing particulièrement utile pour le forecasting et la préparation opérationnelle.

Distinction avec le stress testing

Le capacity testing ne consiste pas à forcer l'échec pour l'échec lui-même.

Il consiste à identifier la charge la plus élevée qui satisfait encore des exigences définies.

- le **capacity testing** s'arrête à la limite acceptable ou près de celle-ci
- le **stress testing** continue au-delà de cette limite pour examiner le comportement en surcharge

La distinction compte parce que de nombreuses décisions business et d'ingénierie dépendent d'une opération sûre, non d'un échec total.

Signification pratique

La capacité n'est pas seulement un nombre.

Elle dépend de :

- mix du workload
- niveau de concurrence
- objectifs de latence
- taux d'erreur acceptable
- contraintes sur les ressources

Pour cette raison, toute valeur de capacité doit toujours être interprétée dans le contexte du workload et des critères d'acceptation utilisés pendant le test.

Interprétation pratique

Le capacity testing est plus utile lorsque l'objectif d'ingénierie est de répondre à :

- « Quel est l'intervalle opérationnel sûr ? »
- « Quelle headroom avons-nous ? »
- « Quand devons-nous scaler ? »
- « Qu'est-ce qui contraint la croissance future ? »

Il est donc l'une des formes de performance testing les plus orientées vers la décision.

Idée clé

Le capacity testing répond à : *« Jusqu'à quel point le système peut-il scaler avant de se dégrader ? »*

Il identifie l'intervalle opérationnel soutenable maximal, et non seulement le point de défaillance.

1.5 – Comportement du système sous charge

Ce chapitre analyse le comportement des systèmes à mesure que la charge de travail (workload) augmente et à l'approche de leurs limites de capacité.

Il se concentre sur les principaux mécanismes pouvant causer une dégradation sous charge, y compris **saturation**, **mise en file**, **perte de throughput** et **amplification de la tail latency**.

Ces concepts sont centraux dans la performance engineering puisqu'ils analysent pourquoi les systèmes peuvent paraître stables à faible charge et devenir instables à proximité de leurs limites de capacité.

Table des matières

1.5.1 Charge vs capacité

Définition

Un système fonctionne sous une charge de travail, mais possède une capacité bien définie.

- **Charge** : la quantité de travail appliquée au système (ex. requêtes par seconde, utilisateurs concurrents)
- **Capacité** : la quantité maximale de travail que le système peut gérer tout en restant stable

Comprendre la relation entre charge et capacité est fondamental en performance engineering.

Elle définit l'enveloppe opérationnelle du système et détermine quand le comportement est prévisible et quand la dégradation commence.

Comportement du système

À faible charge :

- les ressources sont sous-utilisées
- le temps de réponse est stable
- le throughput augmente linéairement avec la charge

À mesure que la charge augmente :

- l'utilisation des ressources croît
- la contention commence à apparaître
- le temps de réponse augmente

Lorsque la charge s'approche de la capacité :

- des files se forment
- la latence augmente rapidement
- le comportement du système devient moins prévisible

Cette transition est l'un des aspects les plus importants de l'analyse des performances.

Un système passe rarement directement de « stable » à « problématique ».

Il traverse généralement une région d'instabilité croissante et d'efficacité réduite.

La capacité n'est pas une valeur fixe

La capacité est souvent comprise à tort comme un ensemble restreint de valeurs.

En réalité, elle dépend de :

- composition du workload (cas d'usage et distribution)
- configuration des ressources (CPU, mémoire, pools)
- état du système (cold vs warm, effets du cache)
- dépendances externes (bases de données, services)

Un système peut gérer :

- 100 req/s pour des requêtes simples
- mais seulement 20 req/s pour des requêtes complexes

La capacité est donc toujours contextuelle.

Elle doit être comprise en relation avec un workload spécifique, un environnement spécifique et des critères d'acceptation.

Capacité effective

La capacité doit être définie sous des contraintes bien précises.

Critères typiques :

- latence dans des limites acceptables (ex. p95)
- taux d'erreur sous le seuil
- utilisation stable des ressources

La charge maximale qui satisfait ces conditions est la **capacité effective**.

C'est cette capacité qui compte du point de vue opérationnel.

Un maximum théorique qui produit une latence inacceptable ou de l'instabilité n'est pas utile dans la pratique.

Implication pratique

La capacité ne peut pas être supposée a priori.

Elle doit être :

- mesurée sous un workload réaliste
- validée par des tests
- surveillée dans le temps

Augmenter la charge au-delà de la capacité effective conduit à :

- dégradation rapide
- comportement instable
- rupture potentielle du système

Cela peut aussi réduire la capacité du système à récupérer rapidement après une surcharge.

Lien avec les concepts précédents

La relation entre charge, latence et concurrence est formalisée par :

→ 1.2.1 Loi de Little

À mesure que la charge augmente :

- la concurrence augmente
- le temps d'attente croît
- le temps de réponse se dégrade

Cette relation constitue l'un des fondements permettant de comprendre le comportement sous charge.

Interprétation pratique

Charge et capacité ne devraient jamais être traitées comme des étiquettes abstraites.

Elles déterminent :

- si le système fonctionne avec de la headroom
- s'il est probable que de la mise en file (queueing) apparaisse
- quelle marge existe avant que l'instabilité apparaisse

En performance engineering, savoir qu'un système « fonctionne » n'est pas suffisant.

Ce qui compte, c'est de savoir dans quelles conditions de charge il reste stable et à quel point il est proche de sa capacité effective.

Idée clé

Un système ne se casse pas lorsqu'il atteint sa capacité.

Il commence à se dégrader avant ce point.

L'objectif de la performance engineering est d'identifier :

- où se trouvent les limites de capacité
- comment le système se comporte à leur proximité
- quelle marge est requise

1.5.2 Saturation et mise en file

Définition

La **saturation** se produit lorsqu'une ressource est occupée la majeure partie du temps ou tout le temps.

La **mise en file** (queueing) se produit lorsque le travail entrant ne peut pas être traité immédiatement et doit être placé en attente : en file.

Ces deux phénomènes sont étroitement corrélés.

Ils figurent parmi les mécanismes les plus importants à la base de la dégradation des performances dans les systèmes réels.

Saturation de la ressource

Une ressource devient saturée lorsque :

- son utilisation s'approche de la limite
- elle a peu ou pas de temps d'inactivité

Exemples typiques :

- CPU proche de 100%
- thread pool entièrement occupé
- connection pool épuisé

À ce stade :

- les nouvelles requêtes ne peuvent pas être traitées immédiatement
- elles doivent attendre

La saturation ne signifie pas nécessairement qu'il y a problème.

Elle signifie que le système a perdu sa marge de traitement et n'est plus en mesure d'absorber du travail supplémentaire sans délai.

Formation de la file

Lorsque les requêtes de travail arrivent plus vite qu'elles ne peuvent être traitées :

- une file se forme
- le temps d'attente augmente

Cela affecte le temps de réponse :

- le temps de service reste le même
- le temps d'attente croît

→ 1.2.3 Temps de service vs temps de réponse

La mise en file est donc la conséquence visible d'une capacité de traitement insuffisante sur une ressource donnée.

Effet non linéaire

La mise en file ne croît pas linéairement.

À mesure que l'utilisation augmente :

- le temps d'attente croît lentement au début
- puis augmente rapidement
- enfin il domine le temps de réponse

De petites augmentations de charge peuvent provoquer de grandes augmentations de latence.

Cela explique pourquoi les systèmes paraissent souvent stables pendant longtemps puis se dégradent brusquement à proximité du seuil de saturation.

Lien avec l'utilisation

L'utilisation joue un rôle central :

→ 1.2.2 Loi dutilisation

Lorsque l'utilisation s'approche de sa limite :

- la probabilité d'attente augmente
- les files croissent
- la latence devient instable

Le point important n'est pas qu'une ressource soit « occupée », mais que lorsqu'elle est occupée de façon persistante, le travail entrant commence à s'accumuler.

Implications pratiques

La mise en file est souvent la cause principale de la dégradation des performances.

Les symptômes incluent :

- augmentation soudaine du temps de réponse

- tail latency élevée (p95, p99)
- files croissantes (threads, connexions, requêtes)

Même si :

- le CPU n'est pas complètement saturé
- la latence moyenne paraît acceptable

la mise en file peut néanmoins être la source dominante du retard.

Cela est particulièrement courant dans les systèmes avec des pools partagés, des opérations bloquantes ou des goulets d'étranglement au niveau des dépendances.

Exemple

Un système traite des requêtes avec :

- temps de service = 10 ms

À faible charge :

- les requêtes sont traitées immédiatement
- temps de réponse ≈ 10 ms

À mesure que la charge augmente :

- les requêtes commencent à attendre
- le temps de réponse devient :
  10 ms (service) + temps d'attente

À forte charge :

- le temps d'attente domine
- le temps de réponse augmente rapidement

Cet exemple vise à illustrer pourquoi la croissance de la latence sous charge est souvent davantage causée par l'attente que par le travail lui-même.

Interprétation pratique

La saturation est la condition.

La mise en file (queueing) est la conséquence.

Le système ne ralentit pas parce que chaque requête exige davantage de calcul, mais parce que davantage de requêtes sont en concurrence pour les mêmes ressources limitées.

Cette distinction est essentielle :

- optimiser le temps de service peut aider
- mais réduire la mise en file est souvent encore plus important

---

Idée clé

La saturation ne casse pas immédiatement le système.

Elle introduit de la mise en file.

La mise en file augmente le temps d'attente.

Le temps d'attente domine le temps de réponse.

C'est le mécanisme principal à la base de la dégradation des performances sous charge.

1.5.3 Dégradation non linéaire

Définition

Les performances du système ne se dégradent pas linéairement à mesure que la charge augmente.

Au contraire, la dégradation suit une évolution non linéaire, en particulier à proximité des limites de capacité.

Cela signifie que la relation entre charge et temps de réponse est souvent d'abord régulière, puis fortement instable près de la saturation.

Comportement linéaire vs non linéaire

À charge faible ou modérée :

- le throughput augmente proportionnellement à la charge
- la latence reste relativement stable

Dans cette région, le système paraît prévisible.

Lorsque la charge s'approche de la capacité :

- de petites augmentations de charge produisent de grandes augmentations de latence
- la variabilité augmente
- le comportement devient instable

Cela marque la transition vers la dégradation non linéaire.

Le système ne se comporte plus de manière proportionnelle à la demande.

Il commence à réagir de manière disproportionnée au travail supplémentaire.

Cause racine

La dégradation non linéaire est principalement causée par :

- effets de mise en file (→ 1.5.2 Saturation et mise en file)
- utilisation élevée des ressources
- contention entre requêtes

À mesure que l'utilisation augmente :

- le temps d'attente croît de manière disproportionnée
- le temps de réponse est dominé par les délais plutôt que par le service

Cela explique pourquoi la dégradation s'accélère souvent brusquement au lieu de croître progressivement.

Effets observables

Les symptômes typiques incluent :

- augmentation rapide de la latence p95 et p99
- élargissement de l'écart entre latence moyenne et tail latency
- augmentation de la variance dans les temps de réponse
- erreurs intermittentes ou timeouts

Ces effets apparaissent souvent soudainement.

Le système peut sembler sain juste avant d'entrer dans une région de grave instabilité.

Intuition trompeuse

Il est courant de supposer :

- « Si le système gère 80 req/s, il devrait gérer 100 req/s avec une latence légèrement plus élevée »

En réalité :

- les performances peuvent rester stables jusqu'à un certain point
- puis se dégrader brutalement au-delà de ce point

Il n'existe souvent pas de transition graduelle.

Cela constitue l'une des erreurs les plus fréquentes dans le capacity planning et dans les attentes de performance.

Exemple

Un système se comporte comme suit :

- jusqu'à 70 req/s → latence stable (~100 ms)
- à 80 req/s → la latence augmente à 150 ms
- à 90 req/s → la latence monte à 400 ms
- à 100 req/s → le système devient instable

La dégradation n'est pas proportionnelle à la charge.

Les derniers incréments de charge ont un effet beaucoup plus important que les précédents.

Implication pratique

Le capacity planning doit tenir compte du comportement non linéaire.

Faire fonctionner un système près de ses limites conduit à :

- latence imprévisible
- performances instables
- mauvaise expérience utilisateur

Les systèmes devraient fonctionner avec une marge de sécurité raisonnable en dessous de la capacité.

Cette marge n'est pas optionnelle.

C'est elle qui permet au système d'absorber la variabilité normale sans entrer dans un comportement instable.

Lien avec les concepts précédents

La dégradation non linéaire est l'effet visible de :

- utilisation croissante (→ 1.2.2 Loi dutilisation)
- mise en file croissante (→ 1.5.2 Saturation et mise en file)

Elle est donc une conséquence au niveau système de mécanismes déjà introduits dans les sections précédentes.

Interprétation pratique

La dégradation non linéaire explique pourquoi les systèmes ne devraient pas être exploités trop près de leur maximum théorique.

Une marge opérationnelle adéquate peut faire la différence entre :

- performances stables

- dégradation imprévisible

Cela explique aussi pourquoi la seule utilisation moyenne des ressources est souvent trompeuse dans l'évaluation de la sécurité en production.

Idée clé

La dégradation des performances n'est pas graduelle.

Elle s'accélère à mesure que le système s'approche de ses propres limites.

Comprendre cette non-linéarité est essentiel pour éviter d'exploiter des systèmes trop près de leurs limites de capacité.

1.5.4 Effondrement du throughput

Définition

L'**effondrement du throughput** se produit lorsque l'augmentation de la charge n'augmente plus le throughput et peut même le réduire.

Au lieu de scaler avec la demande, le système devient moins efficace à mesure que la charge augmente.

C'est l'un des signes les plus clairs que le système fonctionne au-delà de sa propre capacité effective.

Comportement attendu vs effondrement

Dans des conditions normales :

- l'augmentation de la charge augmente le throughput
- jusqu'à ce que le système s'approche des limites de capacité

Cependant, au-delà d'un certain point :

- le throughput cesse d'augmenter
- peut se stabiliser ou diminuer
- la latence augmente significativement

C'est ce qu'on appelle l'effondrement du throughput.

Davantage de travail entrant ne se traduit pas par autant de travail achevé.

Causes racines

L'effondrement du throughput est typiquement causé par :

- mise en file excessive
- contention sur des ressources partagées
- thrashing des ressources (CPU, mémoire, I/O)
- amplification des retries
- scheduling ou locking inefficients

Lorsque le système entre en surcharge :

- davantage de temps est dépensé à gérer la contention qu'à effectuer du travail utile
- la capacité de traitement effective diminue

C'est la raison clé pour laquelle une demande plus élevée peut produire un output plus faible.

Contribution de la mise en file

Lorsque les files croissent :

- les requêtes attendent plus longtemps
- les ressources du système restent occupées
- les nouvelles requêtes ajoutent de la pression sans augmenter le travail achevé

La mise en file peut donc :

- augmenter la latence
- réduire le throughput effectif

Cela est particulièrement visible lorsque le système passe de plus en plus de temps à gérer l'arriéré au lieu de faire de réels progrès.

Contention et thrashing

À forte charge :

- les threads sont en concurrence pour des ressources partagées
- les locks deviennent des hotspots
- le context switching augmente
- la localité du cache se dégrade

Dans des cas extrêmes :

- le système passe plus de temps à coordonner qu'à traiter

Cela conduit à une réduction du throughput.

Le système reste actif, mais son activité devient de plus en plus improductive.

Amplification des retries

Les défaillances sous charge déclenchent souvent des retries.

Cela crée une charge supplémentaire :

- les requêtes ayant échoué sont retentées
- davantage de travail est généré
- la pression augmente encore

Cette boucle de rétroaction peut :

- accélérer l'effondrement
- rendre la récupération difficile

Le comportement des retries n'est donc pas seulement une réponse aux symptômes, mais aussi une cause fréquente de l'aggravation de la surcharge.

Effets observables

Les symptômes typiques incluent :

- throughput qui se stabilise ou diminue malgré l'augmentation de la charge
- forte augmentation de la latence
- augmentation des taux d'erreur (timeouts, 5xx)
- comportement instable ou oscillant

À ce stade, le système peut paraître occupé mais il ne scale plus de manière utile.

Exemple

Un système se comporte comme suit :

- 50 req/s → 50 req/s de throughput
- 80 req/s → 80 req/s de throughput
- 100 req/s → 90 req/s de throughput
- 120 req/s → 70 req/s de throughput

L'augmentation de la charge réduit le throughput effectif.

C'est un indicateur direct du fait que la surcharge est en train « d'endommager » le travail utile.

Implication pratique

L'effondrement du throughput indique que le système fonctionne au-delà de sa propre capacité effective.

À ce point :

- ajouter davantage de charge aggrave les performances
- le système peut devenir instable

La mitigation exige :

- réduire la charge
- supprimer les goulets d'étranglement
- améliorer l'efficacité des ressources

Dans de nombreux cas, la première action corrective n'est pas l'optimisation mais la protection : rate limiting, admission control ou contrôle des retries.

Lien avec les concepts précédents

L'effondrement du throughput est le résultat de :

- dégradation non linéaire (→ 3.5.3 Dégradation non linéaire)
- saturation et mise en file (→ 3.5.2 Saturation et mise en file)

Il peut donc être compris comme un stade avancé du comportement en surcharge.

Interprétation pratique

Un système ne traite pas toujours davantage de travail lorsqu'on lui en applique davantage.

À un certain point, le travail supplémentaire devient destructeur plutôt que productif.

Reconnaître cette transition est essentiel en performance engineering, parce qu'elle marque la différence entre charge élevée et surcharge.

Idée clé

Au-delà d'un certain point, la charge supplémentaire réduit la capacité du système à traiter les requêtes.

Comprendre l'effondrement du throughput est essentiel pour éviter des conditions de surcharge.

1.5.5 Amplification de la tail latency

Définition

L'**amplification de la tail latency** se réfère à l'augmentation disproportionnée des temps de réponse à haut percentile (ex. p95, p99) sous charge.

Alors que la latence moyenne peut paraître acceptable, un sous-ensemble de requêtes devient significativement plus lent.

Cet effet constitue l'un des indicateurs les plus importants d'une expérience utilisateur dégradée et d'une instabilité cachée.

Percentiles vs moyenne

La latence moyenne masque la variabilité.

Les percentiles révèlent la distribution :

- p50 représente la requête typique
- p95 et p99 représentent les requêtes les plus lentes

Sous charge :

- la latence moyenne peut augmenter modérément
- la tail latency peut augmenter drastiquement

→ 1.2.7 Percentiles

Pour cette raison, les seules moyennes ne suffisent pas pour évaluer la qualité réelle des performances.

Causes racines

L'amplification de la tail latency est principalement pilotée par :

- retards de mise en file
- contention sur des ressources partagées
- distribution hétérogène du workload
- variabilité des dépendances (ex. base de données, services externes)

Même de petits retards dans certains composants peuvent :

- se propager à travers le système
- amplifier la latence end-to-end

La tail latency est donc souvent un effet émergent, pas seulement local.

Effet dans les systèmes distribués

Dans les systèmes comportant plusieurs composants :

- une requête dépend souvent de plusieurs services
- la latence globale dépend du composant le plus lent

À mesure que le nombre de dépendances augmente :

- la probabilité d'une requête lente augmente
- la tail latency devient plus marquée

C'est l'une des raisons pour lesquelles la tail latency est particulièrement importante dans les architectures distribuées.

Sous charge

À mesure que la charge augmente :

- les files croissent
- la contention augmente
- la variabilité s'élargit

Cela conduit à :

- un élargissement de l'écart entre moyenne et p95/p99
- des temps de réponse imprévisibles pour un sous-ensemble d'utilisateurs

Le système peut donc paraître globalement stable tout en produisant malgré tout une expérience inacceptable pour une fraction significative des requêtes.

Effets observables

Les symptômes typiques incluent :

- latence moyenne stable avec p95/p99 dégradés
- réponses lentes intermittentes
- timeouts qui n'affectent qu'une fraction des requêtes

Cela peut être trompeur :

- le système paraît « globalement correct »
- mais l'expérience utilisateur est dégradée

Cela explique pourquoi les métriques de queue sont essentielles dans le performance testing et dans la surveillance en production.

Exemple

Un système montre :

- latence moyenne = 120 ms
- latence p95 = 180 ms (acceptable)
- latence p99 = 1200 ms (problématique)

La majorité des requêtes est rapide, mais un petit pourcentage est très lent.

Dans de nombreux systèmes orientés utilisateur, ce petit pourcentage suffit à créer une insatisfaction visible ou des violations des SLO.

Implication pratique

L'évaluation des performances doit prendre en compte la **tail latency**.

S'appuyer sur les moyennes peut :

- masquer des problèmes critiques
- sous-estimer l'impact sur les utilisateurs

Les systèmes devraient être conçus et testés pour :

- contrôler le comportement de queue
- limiter la variabilité sous charge

Cela est particulièrement important pour les systèmes distribués, les API et les applications interactives.

Lien avec les concepts précédents

L'amplification de la tail latency est une conséquence de :

- mise en file (→ 1.5.2 Saturation et mise en file)
- dégradation non linéaire (→ 1.5.3 Dégradation non linéaire)
- interactions et dépendances système

Elle est donc l'une des manifestations les plus visibles du stress du système sous charge.

Interprétation pratique

Les performances ne sont pas définies par la requête moyenne.

Elles sont définies par la prévisibilité des temps de réponse, en particulier pour les requêtes les plus lentes.

Un système avec une latence moyenne acceptable mais un comportement p95/p99 médiocre n'est pas réellement stable du point de vue utilisateur ou opérationnel.

Idée clé

Les performances ne sont pas définies par la requête moyenne.

Elles sont définies par la manière dont le système se comporte pour les requêtes les plus lentes.

Contrôler la tail latency est essentiel pour des systèmes prévisibles et fiables.

◀ 01-04-types-of-performance-tests | ▲ Index | 1.6 – Concurrence et parallélisme ▶

1.6 – Concurrence et parallélisme

Ce chapitre introduit la concurrence et le parallélisme comme concepts fondamentaux dans la performance engineering des systèmes et des applications.

Il introduit le scheduling du travail, la manière dont interagissent des tâches multiples et pourquoi l'overhead de coordination, la contention et la synchronisation deviennent souvent des facteurs limitants sous charge.

La concurrence et le parallélisme sont essentiels pour la scalabilité, mais ils introduisent aussi de la complexité, de l'overhead et des points de rupture qui influencent directement la latence, le throughput et la stabilité du système.

Table des matières

1.6.1 Concurrence vs parallélisme

Définition

Concurrence et **parallélisme** sont des concepts corrélés mais distincts.

Ils sont souvent confondus, mais décrivent des aspects différents du comportement du système.

Comprendre la distinction est essentiel parce qu'un système peut gérer de nombreuses activités simultanément d'un point de vue structurel sans réellement exécuter de nombreuses activités simultanément au niveau matériel.

Concurrence

La **concurrence** se réfère à la capacité d'un système à gérer plusieurs tâches pendant un même intervalle de temps.

Ces tâches :

- peuvent ne pas être exécutées exactement au même moment
- peuvent être « interleaved »
- partagent des ressources système

La concurrence concerne :

- structure
- coordination
- gestion de plusieurs opérations « in flight »

Elle s'intéresse donc principalement à la manière dont le travail est organisé et schedulé.

Parallélisme

Le **parallélisme** se réfère à l'exécution de plusieurs tâches au même instant.

Cela requiert :

- plusieurs unités de traitement (ex. cœurs CPU)
- une véritable exécution simultanée

Le parallélisme concerne :

- exécution
- utilisation du matériel
- accomplir davantage de travail au même instant

Il s'intéresse donc principalement à l'exécution simultanée.

Différence clé

- **Concurrence** = gérer de nombreuses tâches
- **Parallélisme** = exécuter de nombreuses tâches simultanément

Un système peut être :

- concurrent mais non parallèle (single core, tâches « interleaved »)
- parallèle mais non hautement concurrent (peu de tâches de longue durée)

Cette distinction compte parce que les propriétés de scalabilité d'un système dépendent non seulement de la quantité de travail existante, mais aussi de la manière dont ce travail est coordonné et schedulé.

Relation avec les performances

La concurrence influe sur :

- combien de requêtes peuvent être en exécution
- comment les ressources sont partagées
- comment la contention apparaît

Le parallélisme influe sur :

- à quelle vitesse le travail peut être exécuté
- avec quelle efficacité le matériel est utilisé

Les deux influencent :

- throughput
- latence
- scalabilité

Dans la pratique, ajouter de la concurrence sans parallélisme suffisant peut augmenter l'attente et la contention, tandis qu'ajouter du parallélisme sans bon contrôle de la concurrence peut gaspiller des ressources ou exposer des problèmes de coordination.

Intuition pratique

Un système concurrent :

- peut accepter de nombreuses requêtes
- peut néanmoins les traiter séquentiellement ou avec un parallélisme limité

Un système parallèle :

- peut traiter plusieurs requêtes au même moment
- mais peut néanmoins souffrir de contention ou d'overhead de coordination

Pour cette raison, la concurrence et le parallélisme ne devraient pas être traités comme automatiquement bénéfiques.

Leur valeur dépend de la manière dont ils interagissent avec le workload, les ressources partagées et les contraintes d'exécution.

Lien avec les concepts précédents

La concurrence augmente :

- le nombre de requêtes in flight (→ 1.2.1 Loi de Little)

Cela conduit à :

- partage des ressources
- mise en file potentielle (→ 1.5.2 Saturation et mise en file)

C'est l'une des principales raisons pour lesquelles la concurrence devient un sujet central en performance engineering et non seulement une question de programmation.

Interprétation pratique

La concurrence est souvent nécessaire pour supporter de nombreuses opérations simultanées, surtout dans les systèmes réseau et pilotés par I/O.

Cependant, la concurrence augmente aussi la probabilité de :

- interactions sur état partagé
- accumulation de files
- contention sur les verrous
- overhead de coordination

Le parallélisme peut augmenter le throughput, mais seulement si un travail réellement utile est exécuté au lieu d'un travail bloqué ou sérialisé.

Idée clé

La concurrence détermine combien de tâches sont actives.

Le parallélisme détermine combien de tâches sont exécutées au même moment.

Les performances dépendent des deux, et de la manière dont ils interagissent avec les ressources du système.

1.6.2 Threads et modèle d'exécution

Définition

Le **modèle d'exécution** définit comment le travail est exécuté à l'intérieur d'un système.

Dans la plupart des systèmes, le travail est réalisé par des **threads**, qui sont exécutés à l'intérieur d'un **processus**.

Le modèle d'exécution détermine comment les requêtes sont mappées sur les unités d'exécution, comment l'attente est gérée et comment les ressources système sont consommées sous charge.

Processus et threads

Un **processus** est un environnement d'exécution isolé :

- il possède son propre espace mémoire

- il contient des ressources (fichiers, sockets, mémoire)

Un **thread** est une unité d'exécution à l'intérieur d'un processus :

- plusieurs threads partagent la même mémoire du processus
- les threads exécutent des tâches en concurrence

Dans la plupart des applications :

- un processus héberge plusieurs threads
- les threads gèrent les requêtes entrantes

Ce modèle à mémoire partagée rend les threads efficaces pour la communication, mais introduit aussi la complexité de l'état partagé.

Threads

Un thread :

- exécute des instructions
- consomme du temps CPU
- peut se bloquer en attente (ex. I/O, locks)

Plusieurs threads permettent à un système de :

- gérer davantage de requêtes
- superposer calcul et attente
- augmenter la concurrence

Cependant, les threads ne sont pas gratuits.

Chaque thread supplémentaire introduit de l'overhead mémoire, de l'overhead de scheduling et de la complexité de coordination.

Cycle de vie du thread

Un thread traverse typiquement différents états :

- **running** (en exécution active)
- **runnable** (prêt à être exécuté, en attente de CPU)
- **waiting** / blocked (en attente d'une ressource ou d'un événement)

Les performances sont influencées par la manière dont les threads se déplacent entre ces états.

Un système avec de nombreux threads en état « runnable » ou « blocked » peut paraître actif, mais accomplir un progrès utile limité.

Comprendre les états des threads est donc essentiel dans le diagnostic des problèmes de concurrence.

Stack et mémoire

Chaque thread possède sa propre **stack** :

- elle mémorise les appels de méthodes et les variables locales
- elle croît et décroît pendant l'exécution

Implications :

- plus de threads → plus grande utilisation mémoire (une stack par thread)
- chaînes d'appels profondes → plus grande utilisation de la stack
- l'épuisement de la stack peut conduire à des ruptures

Cela est particulièrement pertinent dans les systèmes à haute concurrence.

Le nombre de threads influence donc non seulement le scheduling, mais aussi l'empreinte mémoire et la stabilité.

Modèles d'exécution

Des systèmes différents utilisent des **modèles d'exécution** différents.

Les modèles communs incluent :

Un thread par requête

Chaque requête est gérée par un thread dédié.

Caractéristiques :

- modèle simple
- facile à comprendre
- les opérations bloquantes sont directes

Limites :

- utilisation mémoire élevée avec beaucoup de threads
- scalabilité limitée sous conditions de forte concurrence

Ce modèle est conceptuellement simple, mais il se comporte souvent mal lorsque la concurrence devient très élevée ou lorsque le blocking est fréquent.

Thread pool

Un nombre fixe de threads gère les requêtes entrantes.

Les requêtes sont mises en file et assignées aux threads disponibles.

Caractéristiques :

- concurrence contrôlée
- overhead réduit par rapport à des threads non limités

Limites :

- mise en file lorsque tous les threads sont occupés
- saturation potentielle du pool

Ce modèle est largement utilisé parce qu'il fournit une utilisation contrôlée des ressources, mais il introduit une file explicite et donc une limite de capacité visible.

Modèle event-driven / asynchrone

Le travail est géré en utilisant des opérations **non bloquantes** et des **event loops**.

Caractéristiques :

- peu de threads peuvent gérer de nombreuses requêtes concurrentes
- efficace pour des workloads I/O-bound

Limites :

- modèle de programmation plus complexe
- requiert une gestion soignée des flux asynchrones

Ce modèle réduit le nombre de threads bloqués, mais déplace la complexité vers la coordination, les callbacks, la gestion de l'état et le design non bloquant.

Perspective Java (exemple)

En Java, un modèle d'exécution commun utilise des thread pools.

Par exemple :

```
ExecutorService executor = Executors.newFixedThreadPool(10);

executor.submit(() -> {
    // task logic
});
```

Les requêtes sont :

- envoyées à une file
- exécutées par un nombre limité de threads

Si tous les threads sont occupés :

- les tâches attendent dans la file
- la latence augmente

Pour une explication détaillée des threads en Java, voir :

→ https://ars-digitale.github.io/java-21-study-guide/en/module-07/threads/

Cet exemple est simple, mais il met en évidence une idée clé : des ressources d'exécution limitées introduisent naturellement de la mise en file lorsque la demande dépasse la capacité de traitement immédiate.

Bloquant vs non bloquant

Les threads peuvent :

- **se bloquer** (attendre I/O, verrous, ressources externes)
- **rester actifs** (travail CPU-bound)

Le blocking réduit la concurrence effective :

- les threads sont occupés mais ne progressent pas
- moins de threads sont disponibles pour du nouveau travail

Les approches non bloquantes visent à :

- réduire l'attente inactive
- améliorer l'utilisation des ressources

La distinction est importante parce qu'un nombre élevé de threads ne signifie pas nécessairement un throughput élevé.

Si les threads passent la majorité du temps en attente, la concurrence est présente, mais l'exécution productive est limitée.

Implications pratiques

Le modèle d'exécution détermine :

- comment la concurrence est gérée
- comment les ressources sont utilisées
- comment la mise en file apparaît

Les effets typiques incluent :

- saturation du thread pool → mise en file des requêtes
- opérations bloquantes → throughput réduit
- trop de threads → overhead de context switching

Le modèle d'exécution détermine aussi où les goulets d'étranglement deviennent visibles : dans les files, dans les pools, dans les threads bloqués ou dans les event loops.

Lien avec les concepts précédents

Le comportement des threads impacte directement :

- mise en file (→ 1.5.2 Saturation et mise en file)
- latence sous charge
- capacité effective du système

Il influence aussi la rapidité avec laquelle un système passe d'un comportement stable à la saturation lorsque la concurrence augmente.

Interprétation pratique

Choisir un modèle d'exécution n'est pas seulement une décision de programmation.

C'est une décision de performance.

Le modèle influe sur :

- consommation mémoire
- overhead de scheduling
- latence en conditions d'attente
- scalabilité sous workload réel

Un design facile à implémenter peut ne pas être celui qui se comporte le mieux sous charge soutenue.

Idée clé

Le modèle d'exécution définit comment le travail est schedulé et traité.

Les threads ne sont pas gratuits.

La manière dont ils sont utilisés détermine :

- quelle quantité de travail peut être gérée
- avec quelle efficacité les ressources sont utilisées
- comment le système se comporte sous charge

1.6.3 Contention et synchronisation

Définition

La **contention** se produit lorsque plusieurs threads entrent en compétition pour la même ressource.

La **synchronisation** est le mécanisme utilisé pour coordonner l'accès aux ressources partagées.

Ces concepts sont centraux pour comprendre la dégradation des performances dans les systèmes concurrents.

Ils relient correction et performances : les mêmes mécanismes qui protègent l'état partagé peuvent aussi devenir la source d'attente et de scalabilité réduite.

Ressources partagées

Dans les systèmes concurrents, les threads partagent souvent des ressources telles que :

- structures mémoire (objets, caches)

- verrous et moniteurs
- thread pools et files
- connexions à la base de données
- canaux I/O

Lorsque l'accès n'est pas coordonné, une **corruption** des données peut se produire.

Lorsque l'accès est coordonné, de la **contention** peut apparaître.

Cela rend la synchronisation nécessaire, mais non gratuite.

Synchronisation

La synchronisation garantit que les ressources partagées sont accessibles de manière sûre.

Les mécanismes communs incluent :

- verrous (mutex, moniteurs)
- sections synchronisées
- sémaphores
- opérations atomiques

La synchronisation garantit la correction, mais introduit de l'overhead.

Cet overhead peut provenir de :

- attente
- sérialisation de l'exécution
- memory barriers supplémentaires
- coûts de coordination entre threads

Contention

La **contention** surgit lorsque plusieurs threads tentent d'accéder simultanément à la même ressource.

Lorsque la contention se produit :

- les threads peuvent se bloquer ou attendre
- l'exécution est retardée
- le throughput est réduit

Plus les threads sont en compétition :

- plus le temps d'attente est grand
- plus le parallélisme effectif est faible

Un système hautement concurrent peut donc se comporter comme un système partiellement sérialisé si une grande partie de son travail dépend des mêmes ressources partagées.

Contention sur les verrous

Une forme commune de contention implique les verrous.

Lorsqu'un thread détient un verrou :

- les autres threads doivent attendre
- une file de threads en attente peut se former

Les effets incluent :

- augmentation de la latence
- réduction du throughput
- goulets d'étranglement potentiels

La contention sur les verrous est particulièrement problématique lorsque les sections critiques sont longues, fréquemment accédées ou placées sur des hot paths d'exécution.

Contention vs utilisation

Une contention élevée peut se produire même lorsque l'utilisation du CPU est modérée.

Par exemple :

- de nombreux threads sont en attente d'un verrou
- le CPU est partiellement inactif
- le système paraît sous-utilisé mais est en réalité contraint

C'est une source commune de diagnostics trompeurs.

Cela explique pourquoi une utilisation faible ou modérée du CPU ne signifie pas nécessairement que le système dispose d'une capacité disponible.

Synchronisation fine-grained vs coarse-grained

La synchronisation peut être :

- **coarse-grained** (peu de verrous, grandes sections critiques)
- **fine-grained** (beaucoup de verrous, sections critiques plus petites)

Compromis :

- **coarse-grained** → plus simple mais plus de contention
- **fine-grained** → plus scalable mais plus complexe

Le choix entre les deux modèles dépend des caractéristiques du workload, des patterns d'accès et du coût de la complexité additionnelle de design.

Perspective Java (exemple)

En Java, la synchronisation peut être implémentée en utilisant des blocs `synchronized` :

```
synchronized (lock) {
    // critical section
}
```

Ou bien des verrous explicites :

```
Lock lock = new ReentrantLock();

lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Si de nombreux threads tentent d'entrer dans la même section critique :

- la contention augmente
- les threads se bloquent
- les performances se dégradent

Cet exemple met en évidence comment un mécanisme de correction peut devenir une contrainte de scalabilité sous charge.

Symptômes de la contention

Les indicateurs typiques incluent :

- augmentation du temps de réponse sous charge
- **faible utilisation CPU avec latence élevée**
- threads dans des états blocked ou waiting
- longues files sur des ressources partagées

Ces symptômes apparaissent souvent avant la saturation totale et peuvent être confondus avec d'autres problèmes de ressources s'ils ne sont pas analysés avec attention.

Implications pratiques

La contention limite la scalabilité.

Même avec :

- CPU suffisant
- mémoire adéquate

Un système peut ne pas scaler si :

- les threads passent du temps en attente au lieu d'être en exécution

Réduire la contention a souvent un impact plus grand que l'optimisation des opérations individuelles.

Cela est particulièrement vrai pour les systèmes dont les performances sont contraintes par l'accès partagé plutôt que par le calcul pur.

Lien avec les concepts précédents

La contention contribue à :

- mise en file (→ 1.5.2 Saturation et mise en file)
- dégradation non linéaire (→ 1.5.3 Dégradation non linéaire)
- effondrement du throughput (→ 1.5.4 Effondrement du throughput)

La contention est donc à la fois un phénomène local de synchronisation et un mécanisme de performance au niveau système.

Interprétation pratique

La concurrence augmente les opportunités de recouvrement utile, mais elle augmente aussi la compétition pour les ressources partagées.

Le défi pratique n'est pas simplement d'ajouter davantage de threads, mais de garantir que la concurrence additionnelle produise du travail utile plutôt que de l'attente additionnelle.

Idée clé

La concurrence introduit la nécessité de synchronisation.

La synchronisation introduit la contention.

La contention limite les performances.

Comprendre et contrôler la contention est essentiel pour des systèmes scalables.

1.6.4 Problèmes communs de concurrence

La concurrence introduit de la complexité.

Lorsque plusieurs threads interagissent, des hypothèses incorrectes ou une mauvaise coordination peuvent conduire à des classes spécifiques de problèmes.

Ces problèmes apparaissent souvent sous charge et peuvent affecter sévèrement performances et correction.

Beaucoup d'entre eux sont difficiles à reproduire dans des tests superficiels parce qu'ils dépendent du timing, du scheduling ou de la pression sur les ressources.

1.6.4.1 Race conditions

Définition

Une **race condition** se produit lorsque plusieurs threads accèdent à des données partagées sans synchronisation adéquate, et que le résultat dépend du timing.

Le résultat n'est donc pas déterministe et peut varier d'une exécution à l'autre.

Exemple

Deux threads mettent à jour un compteur partagé :

- Thread A lit valeur = 10
- Thread B lit valeur = 10
- Thread A écrit 11
- Thread B écrit 11

Résultat attendu : 12
Résultat réel : 11

La valeur finale dépend de l'ordre dans lequel des opérations non synchronisées sont exécutées.

Impact

- résultats incorrects
- état du système incohérent
- bugs difficiles à reproduire

Les race conditions peuvent aussi corrompre des hypothèses internes de manières qui n'apparaissent que plus tard sous charge.

Pertinence en performance

Les race conditions peuvent ne pas toujours provoquer d'erreurs visibles, mais :

- elles requièrent souvent de la synchronisation additionnelle
- des correctifs impropres peuvent introduire de la contention

C'est l'une des raisons pour lesquelles correction et performances ne peuvent pas être traitées comme des questions complètement séparées dans les systèmes concurrents.

1.6.4.2 Deadlock

Définition

Un **deadlock** se produit lorsque deux ou plusieurs threads s'attendent indéfiniment les uns les autres.

Chaque thread détient une ressource et attend une autre ressource détenue par l'autre thread.

En conséquence, le progrès s'arrête complètement.

Exemple

- Thread A détient le verrou L1 et attend L2
- Thread B détient le verrou L2 et attend L1

Aucun des deux ne peut progresser davantage.

Ce pattern d'attente circulaire est la caractéristique distinctive du deadlock.

Impact

- le système se bloque
- les requêtes ne sont jamais complétées
- les ressources restent bloquées

Les deadlocks sont particulièrement graves parce qu'ils transforment des ressources actives en ressources bloquées de manière permanente.

Détection

- les threads restent bloqués
- les thread dumps montrent une attente circulaire

Les deadlocks sont souvent détectés via l'analyse des threads plutôt que via des métriques de performance générales.

1.6.4.3 Livelock

Définition

Un **livelock** se produit lorsque les threads ne sont pas bloqués mais changent continuellement d'état en réponse les uns aux autres sans faire de progrès.

Contrairement au deadlock, l'activité continue, mais pas le travail utile.

Exemple

Deux threads retentent de manière répétée une opération :

- les deux détectent un conflit
- les deux retentent au même moment
- le conflit persiste

Le système reste actif, mais le comportement conflictuel continue indéfiniment.

Impact

- le CPU est utilisé
- aucun travail utile n'est complété

Les livelocks peuvent donc ressembler à du traitement actif même si le progrès effectif est égal à zéro.

1.6.4.4 Starvation

Définition

La **starvation** se produit lorsque certains threads n'arrivent pas à obtenir des ressources pendant une période prolongée.

D'autres threads continuent à s'exécuter tandis que certains sont de fait ignorés.

Cela signifie que le système est en train d'opérer du progrès, mais pas d'une manière équitable ou prévisible pour tout le travail.

Causes

- scheduling non équitable
- threads à haute priorité qui dominent l'exécution
- monopolisation des ressources

La starvation est particulièrement problématique lorsqu'un sous-ensemble de requêtes subit une latence extrême tandis que le reste du système paraît fonctionnel.

Impact

- certaines requêtes subissent une latence très élevée
- le système paraît partiellement fonctionnel
- la tail latency augmente

Cela rend la starvation particulièrement pertinente tant du point de vue des performances que de celui de l'expérience utilisateur.

1.6.4.5 Épuisement du thread pool

Définition

L'**épuisement du thread pool** se produit lorsque tous les threads d'un pool sont occupés et que les tâches entrantes doivent attendre.

C'est l'un des goulets d'étranglement liés à la concurrence les plus communs dans les systèmes réels.

Causes

- opérations bloquantes à l'intérieur des threads
- taille insuffisante du pool
- tâches de longue durée

Ces causes peuvent exister indépendamment ou se renforcer mutuellement sous charge croissante.

Effets

- la file des requêtes croît
- la latence augmente
- le throughput peut se dégrader

Si la saturation continue, l'épuisement du thread pool peut aussi contribuer à des timeouts, des retries et de l'instabilité dans les composants upstream.

Lien avec les concepts précédents

L'épuisement du thread pool est un exemple direct de :

- saturation (→ 1.5.2 Saturation et mise en file)
- dégradation non linéaire (→ 1.5.3 Dégradation non linéaire)

Il constitue donc l'une des expressions pratiques les plus claires des comportements système introduits dans le chapitre précédent.

Idée clé

Les problèmes de concurrence ne sont pas seulement des problèmes de correction.

Ce sont aussi des problèmes de performance.

De nombreuses dégradations de performance sont causées par :

- contention
- blocking
- défaillances de coordination

Comprendre ces problèmes est essentiel pour diagnostiquer des systèmes réels.

◀ 1.5 – Comportement du système sous charge | ▲ Index | 01-07-runtime-and-memory-model ▶

1.7 – Runtime et modèle mémoire

Ce chapitre explique comment les "managed runtime" organisent la mémoire, allouent les objets, récupèrent la mémoire qui n'est plus utilisée et se comportent dans une situation de mémoire "sous pression".

On se concentre sur les mécanismes de runtime et de mémoire qui influencent directement la latence, la stabilité et le throughput sous charge.

Comprendre ces mécanismes est essentiel parce que de nombreux problèmes de performance ne sont pas causés seulement par des limites de CPU ou d'I/O, mais par la manière dont la mémoire est allouée, maintenue et récupérée dans le temps.

Table des matières

1.7.1 Structure de la mémoire (heap, stack)

Modèles de gestion de la mémoire

Des systèmes différents utilisent des stratégies de gestion de la mémoire différentes.

Deux approches communes sont :

- **gestion manuelle de la mémoire**
  La mémoire est allouée et libérée explicitement par le programmeur (ex. C, C++)
- **mémoire gérée**
  La mémoire est allouée automatiquement et récupérée par le runtime (ex. Java, .NET)

Ce guide se concentre sur les **systèmes à mémoire gérée**, où :

- les objets sont alloués dynamiquement
- la mémoire est récupérée automatiquement par un ou plusieurs threads dédiés des machines virtuelles respectives (garbage collection)

Cette distinction est importante parce que le comportement des performances change significativement selon que le cycle de vie de la mémoire est contrôlé directement par le programmeur ou indirectement par le runtime.

Définition

La mémoire est organisée en différentes régions avec des rôles bien distincts.

Les deux zones les plus importantes pour le discours sur les performances sont :

- **heap**
- **stack**

Ces deux régions supportent des aspects différents de l'exécution du programme et ont des implications de performance très différentes.

Heap

Le heap est une zone de mémoire partagée utilisée pour l'allocation dynamique.

Dans les runtimes gérés (comme Java) :

- les objets sont alloués sur le heap
- la mémoire est gérée par le runtime
- la garbage collection récupère les objets non utilisés

Implications :

- l'utilisation de la mémoire croît avec le taux d'allocation
- la garbage collection impacte les performances
- l'accès partagé peut introduire de la contention

Le heap n'est donc pas seulement une zone de stockage, mais une section centrale par rapport au comportement du runtime sous charge.

Stack

Chaque thread a son propre stack.

Le stack mémorise :

- les appels de méthode (call frame)
- les variables locales
- les valeurs intermédiaires

Caractéristiques :

- privé pour chaque thread
- croît et diminue pendant l'exécution
- typiquement beaucoup plus petit que le heap

Puisque le stack est privé au thread, l'accès est simple et efficace, mais le nombre de threads influe directement sur l'utilisation totale de la mémoire du stack.

Heap vs stack

Aspect	Heap	Stack
Scope	Partagé entre threads	Privé par thread
Allocation	Dynamique (objets)	Automatique (appels méthode)
Durée	Gérée par le runtime	Liée à l'exécution méthode
Performance	Plus complexe	Très rapide
Impact mémoire	Global	Par thread

Interaction avec les threads

Chaque thread :

- a son propre stack
- partage le heap

Cela crée un modèle dans lequel :

- l'exécution est isolée par thread (stack)
- les données sont partagées entre threads (heap)

Cette interaction est une source de :

- contention (objets partagés)
- overhead de coordination

Cela explique aussi pourquoi la concurrence et le comportement au niveau de la mémoire sont étroitement corrélés dans les systèmes gérés par le runtime.

Implications sur les performances

Heap :

- allocation excessive → augmentation de l'activité GC
- heap grand → cycles de garbage collection plus longs
- accès partagé → contention potentielle

Stack :

- beaucoup de threads → plus grande utilisation totale de la mémoire (un stack par thread)
- chaînes d'appels profondes → augmentation de l'utilisation du stack
- stack overflow → échec dans des cas extrêmes

Ces implications deviennent particulièrement importantes lorsque le système est sous charge soutenue ou à haute concurrence.

Interprétation pratique

Heap et stack ne sont pas seulement des détails d'implémentation.

Ils influencent :

- la manière dont les données sont partagées
- la manière dont le travail est exécuté
- la manière dont la mémoire croît sous concurrence
- l'endroit où apparaît l'overhead du runtime

Un système avec beaucoup de threads et des allocations fréquentes stresse les deux régions de manière différente : le stack à travers le nombre de threads et la profondeur des appels, le heap à travers la création

et la rétention des objets.

Idée clé

Le heap mémorise des données partagées.

Le stack supporte l'exécution.

Les performances dépendent de la manière dont ces deux éléments interagissent sous charge.

Lien avec les concepts précédents

Le comportement de la mémoire impacte directement :

- l'exécution des threads (→ 1.6.2 Threads and execution model)
- la contention (→ 1.6.3 Contention and synchronization)
- la latence sous charge (→ 1.5 System behavior under load)

Pour cette raison le modèle de runtime et de mémoire ne peut pas être analysé séparément de la concurrence et du comportement du système.

1.7.2 Allocation et cycle de vie des objets

Définition

Dans les systèmes à mémoire gérée, les objets sont créés dynamiquement et vivent pendant une certaine période de temps avant d'être récupérés par le runtime.

La manière dont les objets sont alloués et combien de temps ils vivent a un impact direct sur les performances.

Le comportement d'allocation n'est donc pas seulement une question de mémoire, mais aussi une question de latence et de stabilité.

Allocation

L'allocation est le processus de création de nouveaux objets en mémoire.

Dans la plupart des runtimes gérés :

- l'allocation se produit sur le heap
- elle est conçue pour être rapide et efficace
- elle se produit très fréquemment dans les applications typiques

Exemples d'allocation :

- création d'objets request
- construction de structures de données
- traitement de résultats intermédiaires

Dans les systèmes à haut throughput, l'allocation est souvent continue et étroitement liée à l'intensité de la charge de travail.

Taux d'allocation

Le **taux d'allocation** est la quantité de mémoire allouée par unité de temps.

C'est un facteur clé de performance.

Un taux d'allocation élevé signifie :

- plus d'objets créés
- plus grand churn mémoire
- plus grande pression sur le runtime

Même si les allocations individuelles sont rapides, de grands volumes impactent le système.

C'est l'une des raisons pour lesquelles "allocation rapide" ne signifie pas automatiquement "faible overhead mémoire".

Cycle de vie des objets

Les objets ne vivent pas tous pendant la même durée.

Des catégories typiques incluent :

- **objets à courte durée de vie**
  créés et écartés rapidement (ex. données temporaires de request)
- **objets à durée de vie moyenne**
  survivent pendant un certain temps pendant le traitement
- **objets à longue durée de vie**
  restent en mémoire pendant des périodes étendues (ex. cache, état partagé)

Comprendre la durée de vie des objets est essentiel pour raisonner sur le comportement de la mémoire.

Cette caractéristique détermine quelle quantité de mémoire reste active dans le temps et comment le runtime doit organiser le travail de récupération.

Patterns d'allocation

Les systèmes réels tendent à montrer des patterns comme :

- beaucoup d'objets à courte durée de vie par request
- objets à longue durée de vie occasionnels
- bursts d'allocation sous charge

Ces patterns déterminent :

- l'utilisation de la mémoire
- le comportement de la garbage collection
- la stabilité des performances

Les patterns d'allocation sont souvent plus informatifs que les événements d'allocation isolés, parce que le runtime réagit au comportement agrégé dans le temps.

Impact sur les performances

L'allocation en elle-même est habituellement rapide.

L'impact principal dérive de :

- l'augmentation de l'utilisation de la mémoire
- la pression sur la garbage collection

Un taux d'allocation élevé peut conduire à :

- des cycles de garbage collection plus fréquents
- une augmentation de la latence
- des pauses imprévisibles

Le point important est que le coût de la mémoire est souvent indirect : le système paie non seulement pour créer des objets, mais pour gérer les conséquences de la création d'un grand nombre d'entre eux.

Sous charge

Avec l'augmentation de la charge :

- plus de requêtes sont traitées
- plus d'objets sont créés
- le taux d'allocation augmente

Cela amplifie :

- la pression mémoire
- l'activité de garbage collection
- la variabilité de la latence

Un système stable à faible charge peut donc devenir sensible à la mémoire avec l'augmentation du volume de requêtes, même si la logique de chaque requête reste inchangée.

Interaction avec la concurrence

L'allocation est souvent exécutée par plusieurs threads.

Cela peut conduire à :

- de la contention sur les structures mémoire
- une augmentation de l'overhead de coordination
- des patterns d'utilisation de la mémoire non uniformes

Dans les systèmes à haute concurrence :

- le taux d'allocation croît avec la concurrence
- la mémoire devient un goulet d'étranglement partagé

C'est l'une des manières dont la concurrence et le comportement de la mémoire se renforcent mutuellement sous charge.

Implications pratiques

Pour raisonner sur les performances il est important de considérer :

- combien d'objets sont créés par request
- combien de temps ils vivent
- comment le taux d'allocation change sous charge

Comprendre l'allocation est essentiel pour :

- expliquer le comportement de la latence
- identifier des goulets d'étranglement
- prévoir les limites du système

Cela aide aussi à distinguer entre des problèmes causés par le calcul et des problèmes causés par le churn mémoire.

Interprétation pratique

L'allocation est souvent invisible au niveau du code parce qu'elle est facile à écrire et généralement peu coûteuse par opération.

Cependant, au niveau du système, l'allocation répétée change la charge de travail du runtime.

Un design qui crée de grandes quantités d'objets temporaires peut fonctionner correctement, mais quand même imposer une pression significative sur le sous-système de la mémoire.

Lien avec les concepts suivants

L'allocation et la durée de vie des objets influencent directement :

- le comportement de la garbage collection (→ section suivante)
- la pression mémoire
- la latence sous charge

Elles constituent donc la base causale des effets de runtime décrits dans le reste de ce chapitre.

Idée clé

Les performances dépendent de la quantité de mémoire qui est allouée et de combien de temps elle est maintenue.

Les patterns d'allocation façonnent le comportement du système sous charge.

1.7.3 Garbage collection (conceptuelle)

Définition

La garbage collection (GC) est le processus à travers lequel un runtime géré récupère la mémoire qui n'est plus en usage.

Au lieu d'exiger une désallocation explicite, le runtime :

- identifie les objets non utilisés
- libère leur mémoire
- rend disponible de l'espace pour de nouvelles allocations

La garbage collection est l'un des mécanismes distinctifs des runtimes gérés et l'une des principales manières dont le comportement de la mémoire devient visible dans l'analyse des performances.

Principe de base

Un objet est éligible pour la "collection" lorsqu'il n'est plus atteignable (pointé) par d'autres éléments du programme.

Cela signifie :

- aucune référence active ne pointe vers lui
- il ne peut pas être accédé par le programme

Le runtime périodiquement :

- scanne les références aux objets
- identifie les objets non atteignables
- récupère leur mémoire

Ce modèle permet une gestion automatique de la mémoire, mais implique aussi que le travail de récupération doive être exécuté pendant l'exécution du programme.

Cycle allocation et récupération

L'utilisation de la mémoire suit un cycle :

1. les objets sont alloués
2. les objets deviennent inutilisés
3. la garbage collection récupère la mémoire

Ce cycle se répète continuellement pendant l'exécution.

Le runtime alterne donc entre allocation de nouvelle mémoire et récupération d'ancienne mémoire, avec un comportement global guidé par le taux d'allocation et par les patterns de rétention.

Perspective Java (exemple)

En Java, l'allocation d'objets est fréquente et économique.

Par exemple :

```
for (int i = 0; i < 1_000_000; i++) {
    String s = new String("test");
}
```

Ce code crée un grand nombre d'objets à courte durée de vie.

Dans un runtime géré :

- ces objets sont alloués rapidement sur le heap
- ils deviennent non atteignables peu après la création
- la garbage collection les récupère

Si de tels patterns d'allocation se vérifient sous charge :

- l'activité GC augmente
- la pression mémoire croît
- la latence peut devenir instable

L'impact dépend non d'une seule allocation, mais du **taux d'allocation dans le temps**.

Pour cette raison le comportement de la mémoire doit être analysé comme un pattern, non comme une opération isolée.

Exemple : rétention des objets

Les objets qui restent référencés ne sont pas collectés.

```
List<String> cache = new ArrayList<>();

while (true) {
    cache.add(new String("data"));
}
```

Dans ce cas :

- les objets sont alloués continuellement
- ils ne sont jamais relâchés
- l'utilisation de la mémoire croît dans le temps

Cela conduit à :

- augmentation de la pression mémoire
- cycles de garbage collection plus coûteux
- instabilité potentielle du système

Cet exemple illustre la différence entre churn temporaire d'allocation et rétention persistante.

Coût de la garbage collection

La garbage collection n'est pas gratuite.

Elle introduit un overhead :

- temps CPU pour analyser la mémoire
- pauses pendant la collecte (selon la stratégie/policy de GC)

Le coût dépend de :

- taux d'allocation
- nombre d'objets actifs
- taille de la mémoire

En d'autres termes, le coût GC dépend non seulement de la quantité de mémoire qui existe, mais de la quantité de mémoire qui est active et encore atteignable.

Effet stop-the-world

Certaines phases (de certaines policies) de la garbage collection peuvent suspendre l'exécution de l'application.

Pendant ces pauses :

- les threads applicatifs sont temporairement en stand-by
- aucun travail applicatif n'est exécuté

Même des pauses brèves peuvent :

- augmenter la latence
- influencer les temps de réponse de queue (p95, p99)

C'est l'une des raisons pour lesquelles les problèmes GC apparaissent souvent d'abord dans l'analyse de la latence basée sur les percentiles plutôt que dans les moyennes.

Comportement générationnel (conceptuel)

La majorité des runtimes modernes utilise une approche générationnelle.

Basée sur l'observation :

- la majorité des objets a une courte durée de vie
- peu d'objets ont une durée de vie prolongée

La mémoire est organisée de telle sorte que :

- les objets à courte durée de vie soient collectés fréquemment
- les objets à longue durée de vie soient collectés moins souvent

Cela améliore l'efficacité parce que récupérer de nombreux objets à courte durée de vie est habituellement plus économique que scanner répétitivement de la mémoire à longue rétention.

Sous charge

Avec l'augmentation de la charge :

- le taux d'allocation augmente
- la garbage collection est exécutée plus fréquemment

Cela peut conduire à :

- plus grande utilisation de la CPU
- pauses plus fréquentes
- augmentation de la variabilité de la latence

Sous charge importante, la GC peut donc passer d'un mécanisme de maintenance en background à une partie visible du comportement des performances du système.

Interaction avec le cycle de vie des objets

Le comportement de la garbage collection dépend de :

- combien d'objets sont créés
- combien de temps ils vivent

Patterns typiques :

- beaucoup d'objets à courte durée de vie → collectes fréquentes
- beaucoup d'objets à longue durée de vie → collectes plus lourdes

Pour cette raison allocation et rétention doivent être analysées ensemble : le nombre d'objets à lui seul n'est pas suffisant.

Effets observables

Les problèmes de garbage collection apparaissent souvent comme :

- pics de latence
- latence de queue (dégradation p95/p99)
- pauses périodiques
- augmentation de l'utilisation CPU sans cause évidente

Ces symptômes sont souvent intermittents, ce qui rend les problèmes liés à la GC difficiles à diagnostiquer sans corréler des signaux de mémoire et de latence.

Implications pratiques

L'analyse des performances doit considérer :

- taux d'allocation
- distribution de la durée de vie des objets
- fréquence et coût des cycles GC

L'optimisation typiquement se concentre sur :

- compréhension des patterns d'allocation
- réduction de la création inutile d'objets
- contrôle de la pression mémoire

Le tuning du collector peut aider, mais habituellement il est plus efficace de comprendre à l'avance pourquoi le runtime est sous pression.

Interprétation pratique

La garbage collection n'est pas un bug ou une anomalie.

C'est un mécanisme nécessaire du runtime.

La question sur les performances n'est pas de savoir si la GC existe, mais si son coût de fonctionnement reste compatible avec la charge de travail et les objectifs de latence du système.

Lien avec les concepts précédents

La garbage collection est directement liée à :

- allocation (→ 1.7.2 Allocation et cycle de vie des objets)
- structure de la mémoire (→ 1.7.1 Structure de la mémoire)
- latence de queue (→ 1.5.5 Tail latency amplification)

Elle est donc à la fois un mécanisme de runtime et un contributeur au niveau système à la variabilité des performances.

Idée clé

La garbage collection permet la gestion automatique de la mémoire mais introduit de la variabilité.

Les performances dépendent de l'efficacité avec laquelle la mémoire est récupérée.

1.7.4 Pression mémoire et performance

Définition

La pression mémoire se réfère au stress placé sur le système de mémoire lorsque allocation, rétention et récupération interagissent sous charge.

Elle ne concerne pas seulement la quantité de mémoire utilisée, mais la manière dont la mémoire est gérée et se comporte dans le temps.

La pression mémoire est donc une condition dynamique, non simplement une mesure statique de l'occupation du heap.

Ce qui crée la pression mémoire

La pression mémoire est guidée par une combinaison de facteurs :

- taux d'allocation élevé
- grand nombre d'objets actifs
- longue durée de vie des objets
- récupération inefficace de la mémoire

Ces facteurs se renforcent mutuellement et déterminent combien de travail le runtime doit accomplir pour maintenir la mémoire utilisable.

Allocation vs rétention

Deux patterns différents peuvent créer de la pression :

- **taux d'allocation élevé**
  de nombreux objets sont créés et rapidement écartés
- **rétention élevée**
  les objets restent en mémoire pendant de longues périodes

Ces patterns créent de la pression de manières différentes.

Un taux d'allocation élevé augmente le churn et la fréquence de collecte.

Une rétention élevée augmente la quantité de mémoire qui reste active et doit être scannée ou préservée.

Exemple : taux d'allocation élevé

```
for (int i = 0; i < 1_000_000; i++) {
    String s = new String("test");
}
```

Caractéristiques :

- beaucoup d'objets à courte durée de vie
- allocation fréquente
- garbage collection fréquente

Effets :

- augmentation de l'activité GC

- overhead CPU
- pics de latence potentiels

Cet exemple met en évidence une pression guidée par le churn plutôt que par la rétention à long terme.

Exemple : rétention de la mémoire

```
List<String> cache = new ArrayList<>();

while (true) {
    cache.add(new String("data"));
}
```

Caractéristiques :

- les objets sont maintenus
- l'utilisation de la mémoire croît continuellement

Effets :

- augmentation de l'utilisation du heap
- cycles de garbage collection plus lourds
- instabilité ou échec final

Cet exemple met en évidence une pression guidée par la mémoire retenue plutôt que par la seule fréquence d'allocation temporaire.

Sous charge

Avec l'augmentation de la charge du système :

- plus de requêtes sont traitées
- plus d'objets sont créés
- plus d'objets sont retenus

Cela conduit à :

- augmentation du taux d'allocation
- augmentation de l'utilisation de la mémoire
- augmentation de l'activité GC

La pression mémoire amplifie :

- la variabilité de la latence
- la latence de queue

Pour cette raison la dégradation liée à la mémoire devient souvent plus visible lorsque le système passe d'une charge modérée à une charge soutenue élevée.

Interaction avec la garbage collection

La garbage collection répond à la pression mémoire.

Sous pression :

- les collectes deviennent plus fréquentes
- les pauses peuvent augmenter
- l'utilisation de la CPU croît

Dans des cas extrêmes :

- la GC domine l'exécution
- le travail utile diminue

Lorsque cela arrive, le runtime est en train de dépenser une part significative de son effort de travail dans la gestion même de la mémoire plutôt que dans le traitement du travail applicatif.

Symptômes observables

La pression mémoire apparaît souvent comme :

- pics de latence sans goulet d'étranglement CPU clair
- dégradation de la latence de queue (p95, p99)
- pauses périodiques
- augmentation de la fréquence GC
- croissance de l'utilisation de la mémoire dans le temps

Ces symptômes sont particulièrement importants parce qu'ils peuvent être pris pour une lenteur générique si le comportement de la mémoire n'est pas analysé directement.

Intuition pratique

Un système peut apparaître :

- légèrement chargé (CPU modérée)
- mais quand même lent

Cela indique souvent :

- pression mémoire
- overhead lié à la GC

C'est l'une des raisons principales pour lesquelles la seule CPU n'est pas suffisante pour évaluer la santé du système.

Modèle simplifié

Le comportement du système peut être approximé comme :

- taux d'allocation ↑ → activité GC ↑
- rétention ↑ → utilisation de la mémoire ↑
- activité GC ↑ → variabilité de la latence ↑

Ces relations ne sont pas linéaires.

Elles dépendent de la stratégie du runtime, de la forme de la charge de travail, de la durée de vie des objets et de la quantité de données actives.

Implications pratiques

Pour gérer la pression mémoire :

- comprendre les patterns d'allocation
- identifier les objets à longue durée de vie
- monitorer le comportement GC
- corréler les métriques mémoire avec la latence

L'optimisation devrait se concentrer sur :

- réduire les allocations non nécessaires
- contrôler la durée de vie des objets
- éviter une rétention non limitée

Dans de nombreux cas, la solution la plus efficace n'est pas le tuning du collector, mais la réduction du travail mémoire que le runtime est forcé d'exécuter.

Lien avec les concepts précédents

La pression mémoire contribue à :

- dégradation non linéaire (→ 1.5.3 Non-linear degradation)
- effondrement du throughput (→ 1.5.4 Throughput collapse)
- amplification de la latence de queue (→ 1.5.5 Tail latency amplification)

Elle est donc un pont direct entre les internes du runtime et le comportement visible du système sous charge.

Interprétation pratique

La pression mémoire explique pourquoi un système peut se dégrader même lorsqu'il n'est pas manifestement limité par la CPU ou bloqué extérieurement.

Un runtime sous stress au niveau de la mémoire peut apparaître actif, mais produire une latence croissante, un throughput réduit et un comportement instable.

Cela fait de la pression mémoire l'une des causes cachées les plus importantes dans la dégradation des performances des runtimes gérés.

Idée clé

La pression mémoire dérive de l'interaction entre allocation, rétention et garbage collection sous charge.

Comprendre cette interaction est essentiel pour expliquer des problèmes de latence et de stabilité dans les systèmes réels.

◀ 1.6 – Concurrence et parallélisme | ▲ Index | 01-08-resource-level-performance ▶

1.8 – Performance au niveau des ressources

Ce chapitre investigue comment les ressources fondamentales du système se comportent sous charge et comment elles peuvent contraindre les performances.

On se concentre sur CPU, I/O, réseau et sur les modalités selon lesquelles des goulots d'étranglement peuvent émerger lorsque l'une des ressources se sature avant les autres.

Comprendre la performance au niveau des ressources est essentiel parce que la dégradation du système est souvent le résultat visible des limites des ressources plutôt que de la seule logique applicative.

Table des matières

1.8.1 Comportement de la CPU

Définition

La **CPU** est le composant responsable de l'exécution des instructions.

Les performances de la CPU sont déterminées non seulement par la rapidité avec laquelle les instructions sont exécutées, mais par la manière dont l'exécution est schedulée entre des charges de travail

concurrentes.

Cette distinction est importante parce que la dégradation liée à la CPU est souvent causée par la pression de scheduling, la mise en file d'attente et les contentions, plutôt que seulement par le coût computationnel.

Utilisation de la CPU vs saturation

L'**utilisation de la CPU** représente quelle part de la capacité de la CPU est utilisée.

Une utilisation élevée n'est pas nécessairement l'indice d'un problème éventuel.

La **saturation de la CPU** se vérifie lorsque :

- il y a plus de travail que la CPU ne peut en exécuter
- les threads sont prêts à exécuter mais ne peuvent pas être schedulés immédiatement

Distinction clé :

- **utilisation élevée** → la CPU est occupée
- **saturation** → la CPU est surchargée

Un système peut donc montrer une utilisation élevée de la CPU et continuer malgré tout à se comporter de manière acceptable, tant que le travail exécutable ne s'accumule pas plus vite que la CPU ne peut le traiter.

Scheduling et run queue

Les threads n'exécutent pas de manière continue.

Ils sont schedulés par le système d'exploitation.

À tout moment :

- certains threads sont en **exécution**
- certains sont **en attente** d'exécuter (run queue)

Lorsque le nombre de threads exécutables dépasse le nombre de cœurs CPU disponibles :

- les threads s'accumulent dans la run queue
- les retards de scheduling augmentent

Cela impacte directement la latence (→ 1.5 System behavior under load) et peut être investigué en utilisant les relations de concurrence (→ 1.2.1 Little's Law (system-level concurrency)).

La run queue est donc un signal critique de pression de la CPU, parce qu'elle montre non seulement que la CPU est occupée, mais qu'il y a du travail qui est en attente d'être exécuté.

Comportement observable (exemple)

Un système sous pression de la CPU montre un nombre croissant de threads exécutables.

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 7  0      0  12000  45000 300000    0    0     2     1 1200 3000 90  8  2  0  0
 8  0      0  11000  45000 300000    0    0     1     2 1300 3200 92  6  2  0  0
```

Interprétation :

- run queue (`r`) élevée → threads en attente de la CPU
- CPU idle (`id`) proche de zéro → aucune capacité disponible
- utilisation de la CPU (`us + sy`) proche de la saturation

Cela indique qu'il y a des threads prêts à exécuter mais qui ne peuvent pas être schedulés immédiatement par manque de cœurs disponibles (→ 1.6 Concurrency and parallelism).

Le point important est que la saturation de la CPU n'est pas définie seulement par des valeurs en pourcentage, mais par la présence de travail exécutable qui ne peut pas progresser immédiatement.

Impact sur les performances

Lorsque la CPU devient saturée :

- les retards de scheduling augmentent
- le temps de réponse augmente
- le throughput peut se stabiliser ou diminuer

Cet effet est non linéaire (→ 1.5.3 Non-linear degradation).

Avec l'augmentation de la saturation de la CPU, l'applicatif peut passer (progressivement) plus de temps à attendre d'être schedulé pour l'exécution plutôt qu'à accomplir un travail utile.

Interaction avec la concurrence

La concurrence augmente le nombre de threads actifs.

Avec la croissance de la concurrence :

- plus de threads entrent en compétition pour la CPU
- la longueur de la run queue augmente
- l'overhead de scheduling augmente

Au-delà d'un certain point :

- ajouter des threads réduit les performances au lieu de les améliorer (→ 1.6 Concurrency and parallelism).

C'est la raison pour laquelle ajouter plus de travail concurrent ne produit pas toujours un meilleur throughput.

Si le temps CPU devient la ressource limitante, la concurrence se transforme en pression de scheduling.

Implications pratiques

Pour raisonner sur le comportement de la CPU :

- distinguer utilisation et saturation
- observer les threads exécutables, pas seulement le %CPU
- corréler les métriques CPU avec la latence (→ 1.2 Core metrics and formulas)

Les problèmes CPU ne concernent souvent pas la pure utilisation, mais la **contention pour l'exécution**.

Il est donc possible qu'un système apparaisse "pleinement occupé" sans être instable, ou bien qu'il apparaisse seulement modérément occupé tout en montrant déjà des retards de scheduling.

Interprétation pratique

L'analyse de la CPU devrait se concentrer sur la capacité du système à suivre le rythme du travail exécutable.

Une CPU occupée n'est pas automatiquement un problème.

Une CPU saturée devient un problème lorsque les tâches exécutables s'accumulent, que la latence augmente et que le throughput ne scale plus avec la demande entrante.

Idée clé

Les performances de la CPU sont limitées par le scheduling.

Lorsque les threads ne peuvent pas être schedulés immédiatement, la latence augmente même si le système apparaît pleinement utilisé.

1.8.2 I/O et disque

Définition

Les **opérations d'I/O** impliquent la lecture depuis ou l'écriture vers des dispositifs de stockage.

Contrairement aux opérations CPU, l'I/O est typiquement plus lent et souvent bloquant.

Cela signifie que beaucoup de problèmes de performance qui impliquent l'I/O sont dominés par le temps d'attente plutôt que par le calcul actif.

Latence vs throughput

Les performances de l'I/O ont deux dimensions clés :

- **latence** → temps pour compléter une seule opération
- **throughput** → nombre d'opérations par unité de temps

Un throughput élevé ne garantit pas une faible latence.

Un système peut déplacer une grande quantité de données globale tandis que les requêtes individuelles expérimentent malgré tout des temps d'attente significatifs.

Comportement bloquant

Beaucoup d'opérations d'I/O sont bloquantes :

- un thread démarre une opération
- il attend jusqu'à son achèvement

Pendant ce temps :

- le thread n'exécute pas de travail utile
- il peut maintenir des ressources (locks, connexions)

C'est l'une des raisons principales pour lesquelles les goulots d'étranglement d'I/O se propagent souvent en pression sur les thread pools, en mise en file d'attente et en réduction de la concurrence effective.

Effets de mise en file d'attente

Lorsque plusieurs requêtes exécutent de l'I/O :

- les opérations se mettent en file au niveau du dispositif
- le temps d'attente augmente

Avec l'augmentation de la longueur de la file :

- la latence augmente
- la variabilité augmente (→ 1.5 System behavior under load)

Cela peut être exprimé comme un retard de mise en file d'attente (→ 1.2.3 Service time vs response time (queueing)).

Le point important est que le coût de l'I/O n'est pas limité à la durée de l'opération en elle-même.

Il inclut aussi le temps passé à attendre que les opérations précédentes soient complétées.

Comportement observable (exemple)

Un système sous pression d'I/O montre des temps d'attente croissants.

```
$ iostat -x 1
Device            r/s     w/s   await   %util
sda               120      80    35.0    95.0
sda               130      90    42.0    98.0
```

Interprétation :

- `await` élevé → les requêtes passent un temps significatif en attente
- `%util` proche de 100% → le dispositif est saturé
- latence croissante indique une accumulation de file

Cela reflète des effets de mise en file d'attente (→ 1.2 Core metrics and formulas).

La valeur `await` croissante est particulièrement importante, parce qu'elle révèle souvent que le dispositif n'est pas simplement occupé, mais de plus en plus incapable d'absorber le travail entrant sans retard additionnel.

Impact sur les performances

Lorsque l'I/O devient un goulot d'étranglement :

- la latence des requêtes augmente
- le throughput peut se dégrader
- les threads passent plus de temps à attendre qu'à exécuter

Cela peut réduire la capacité effective du système même lorsque l'utilisation de la CPU reste modérée.

Un système peut donc être limité par l'I/O sans apparaître limité par la CPU.

Interaction avec la concurrence

Plus de requêtes concurrentes conduisent à :

- plus d'opérations d'I/O
- des files sur le dispositif plus longues
- une latence augmentée

Augmenter la concurrence n'améliore pas les performances si le dispositif est saturé (→ 1.6 Concurrency and parallelism).

Au-delà d'un certain point, une concurrence additionnelle augmente seulement l'attente et aggrave le temps de réponse.

Implications pratiques

Pour raisonner sur le comportement de l'I/O :

- se concentrer sur la latence (`await`), pas seulement sur le throughput
- identifier l'accumulation de file
- corréler l'attente d'I/O avec la latence applicative (→ 1.5 System behavior under load)

Les problèmes d'I/O sont souvent mal compris parce que le throughput peut rester acceptable tandis que la latence se dégrade significativement.

Interprétation pratique

Les performances de l'I/O devraient être évaluées comme un système d'attente.

La question centrale n'est pas seulement combien d'opérations par seconde le dispositif peut supporter, mais pendant combien de temps les opérations attendent lorsque la charge de travail s'intensifie.

Un sous-système de stockage qui se comporte bien à faible concurrence peut se dégrader brutalement lorsque les requêtes commencent à s'accumuler.

Idée clé

Les performances de l'I/O sont dominées par le temps d'attente.

Lorsque les files croissent, la latence augmente et la réactivité du système se dégrade.

1.8.3 Comportement du réseau

Définition

Les performances du **réseau** sont déterminées par le transfert de données entre systèmes.

Elles dépendent à la fois de la latence et de la largeur de bande.

Dans les systèmes distribués, le comportement du réseau est souvent un contributeur principal au temps de réponse end-to-end, spécialement lorsque les requêtes traversent plusieurs services.

Latence et round trip

La communication réseau requiert souvent des échanges multiples.

Chaque échange introduit :

- retard de transmission
- retard de propagation
- retard de traitement

Des round trips multiples amplifient la latence totale (→ 1.5 System behavior under load).

Cela est particulièrement important dans les chaînes de requêtes dans lesquelles chaque appel de service dépend de la réponse du précédent.

Même de petits retards peuvent s'accumuler significativement à travers de multiples hops réseau.

Limitations de largeur de bande

La largeur de bande définit la quantité de données qui peuvent être transférées par unité de temps.

Lorsque la largeur de bande est limitée :

- des payloads grands demandent plus de temps pour être transférés
- le throughput devient contraint

La largeur de bande compte donc surtout lorsque la quantité de données transférées devient suffisamment grande pour dominer le temps de communication.

La latence, au contraire, compte aussi pour des payloads petits lorsque beaucoup de round trips sont requis.

Amplification sous charge

Avec l'augmentation de la charge :

- plus de requêtes sont envoyées sur le réseau
- la contention augmente

- des files peuvent se former dans les buffers

Cela conduit à :

- augmentation de la latence
- retards de paquets ou retransmissions (→ 1.5.5 Tail latency amplification)

Sous charge, la variabilité du réseau devient particulièrement importante parce que des retards occasionnels peuvent influencer seulement une partie du trafic tout en dégradant malgré tout l'expérience utilisateur globale.

Comportement observable (exemple)

Un système sous pression réseau montre une accumulation de connexions et de files.

```
$ ss -s
Total: 1200
TCP:   900 (estab 850, timewait 30)

Transport Total     IP        IPv6
*         1200      -         -
RAW       0         0         0
UDP       50        40        10
TCP       870       800       70
```

Interprétation :

- grand nombre de connexions établies → haute concurrence
- accumulation de connexions peut indiquer un traitement lent ou des retards réseau

Un nombre croissant de connexions ouvertes peut indiquer que les requêtes ne se complètent pas assez rapidement, soit parce que les services downstream sont lents, soit parce que le système n'est pas capable de traiter efficacement le travail réseau.

Impact sur les performances

Les contraintes réseau conduisent à :

- augmentation du temps de réponse
- plus grande variabilité
- retards en cascade entre services

Dans les architectures distribuées, ces retards se propagent et s'amplifient souvent parce qu'une seule interaction réseau lente peut retarder de nombreuses opérations dépendantes.

Interaction avec le design du système

Les systèmes distribués amplifient les effets du réseau :

- plusieurs services introduisent plusieurs hops réseau
- la latence s'accumule à travers les appels (→ 1.5 System behavior under load)

Un système avec beaucoup de frontières de service peut donc souffrir d'une latence induite par le réseau même lorsque chaque appel individuel apparaît relativement peu coûteux.

Implications pratiques

Pour raisonner sur le comportement du réseau :

- considérer le nombre de round trips
- observer les patterns de connexion
- corréler l'activité réseau avec la latence

Il est aussi important de distinguer entre :

- comportement limité par la largeur de bande
- comportement limité par la latence
- retard induit par les dépendances

Ce sont des problèmes corrélés mais non identiques.

Interprétation pratique

Les performances du réseau ne concernent pas seulement la vitesse à laquelle les bytes se déplacent.

Elles concernent aussi la fréquence à laquelle les systèmes communiquent, combien de dépendances sont impliquées et comment les retards dans un composant influencent les autres.

Dans beaucoup d'architectures de services, réduire des round trips non nécessaires peut améliorer la latence plus efficacement qu'augmenter simplement la largeur de bande.

Idée clé

Les performances du réseau sont guidées par la latence et par les patterns de communication.

Sous charge, de petits retards s'accumulent et impactent significativement le temps de réponse.

1.8.4 Saturation des ressources et goulots d'étranglement

Définition

Un **goulot d'étranglement** (bottleneck) est la ressource qui limite les performances du système.

La saturation se vérifie lorsque cette ressource opère à pleine capacité ou dans des intervalles proches de sa capacité limite.

C'est le point auquel une charge de travail additionnelle ne se traduit plus par un throughput utile proportionnel.

Identifier la ressource limitante

À tout moment, les performances du système sont contraintes par une ressource dominante :

- CPU
- I/O
- réseau
- mémoire (indirectement via GC → 1.7 Runtime and memory model)

Identifier cette ressource est essentiel.

Sans identifier la réelle ressource limitante, les efforts d'optimisation ciblent souvent les symptômes plutôt que les causes.

Principe du goulot d'étranglement unique

Même dans les systèmes complexes :

- les performances sont typiquement limitées par une contrainte primaire

Améliorer des ressources non limitantes a peu d'effet.

Ce principe est l'une des raisons pour lesquelles la performance engineering doit rester orientée système.

Beaucoup de ressources peuvent apparaître actives, mais une seule, généralement, détermine la limite de capacité courante.

Effets en cascade

Lorsqu'une ressource devient saturée :

- les files s'accumulent
- la latence augmente
- les composants upstream ralentissent

Cela peut se propager à travers le système (→ 1.5 System behavior under load).

Un goulot d'étranglement local peut donc devenir un problème étendu à l'ensemble du système, puisque les retards se diffusent vers appelants, workers, pools et services dépendants.

Interaction entre les ressources

Les ressources ne sont pas indépendantes :

- un I/O lent augmente le temps d'attente des threads → influence le scheduling de la CPU (→ 1.8.1 CPU behavior)
- les retards réseau augmentent la durée des requêtes → augmentent l'utilisation de la mémoire (→ 1.7 Runtime and memory model)
- la saturation de la CPU retarde le traitement → augmente la taille des files (→ 1.2.1 Little's Law (system-level concurrency))

Cette interaction explique pourquoi les goulots d'étranglement se déplacent souvent ou apparaissent couplés avec la variation des conditions de charge de travail.

Le facteur limitant peut changer lorsqu'une partie du système est améliorée ou lorsque la composition de la charge de travail change.

Patterns observables

Signes communs de goulots d'étranglement :

- CPU proche de la saturation avec run queue élevée
- latence I/O en augmentation avec utilisation élevée du dispositif
- retards réseau avec comptage de connexions croissant

Ces patterns sont utiles parce qu'ils relient les symptômes au niveau du système avec des comportements spécifiques des ressources.

Ils aident à réduire l'ambiguïté diagnostique.

Impact sur le comportement du système

Lorsqu'un goulot d'étranglement est atteint :

- le throughput cesse d'augmenter
- la latence croît rapidement
- le système devient instable sous charge supplémentaire

Cela correspond à :

- dégradation non linéaire (→ 1.5.3 Non-linear degradation)
- effondrement du throughput (→ 1.5.4 Throughput collapse)

À ce stade, une demande additionnelle aggrave souvent la situation au lieu d'augmenter l'output utile.

Implications pratiques

Pour analyser les performances :

- identifier la ressource saturée
- corréler les métriques de ressource avec la latence
- concentrer l'optimisation sur le facteur limitant

Un diagnostic correct dépend donc de la compréhension non seulement de quelles ressources sont occupées, mais de laquelle d'entre elles est en train de déterminer actuellement le comportement de l'ensemble du système.

Interprétation pratique

L'analyse des goulots d'étranglement est le pont entre observation et action.

Le but n'est pas simplement de collecter des métriques de CPU, d'I/O ou de réseau, mais de déterminer quelle ressource contraint le travail utile au point opérationnel courant.

Une fois cette ressource identifiée, l'optimisation devient significative.

Idée clé

Les performances du système sont limitées par son goulot d'étranglement.

Comprendre quelle ressource est saturée est essentiel pour expliquer et améliorer le comportement sous charge.

◀ 01-07-runtime-and-memory-model | ▲ Index | 01-09-common-performance-problems ▶

1.9 – Problèmes communs de performance

Ce chapitre décrit des problèmes communs de performance qui apparaissent dans les systèmes réels sous charge.

Ces problèmes n'appartiennent pas à des catégories isolées. Ils interagissent souvent, se renforcent mutuellement et deviennent visibles sous la forme d'une croissance de la latence, d'une perte de throughput, d'une instabilité ou d'une dégradation en queue.

Le but de ce chapitre est de relier des symptômes récurrents aux mécanismes sous-jacents déjà introduits dans les chapitres précédents.

Table des matières

1.9.1 Inefficacité CPU-bound

Définition

Une inefficacité CPU-bound se vérifie lorsque le système dépense un temps CPU excessif en accomplissant un travail qui pourrait être réduit, optimisé ou même évité.

Cela ne signifie pas nécessairement que le système soit toujours CPU-saturé.

Cela signifie que le temps CPU disponible est consommé de manière inefficace, réduisant la quantité de travail utile que le système peut accomplir avant d'atteindre la saturation.

Causes typiques

- algorithmes inefficients (ex. complexité non nécessaire)
- calculs répétés
- absence de caching pour des opérations coûteuses
- transformations de données excessives

Ces causes sont communes parce que l'inefficacité CPU émerge souvent d'un code fonctionnellement correct mais structurellement coûteux.

En performance engineering, l'inefficacité est davantage impactante lorsqu'elle se constate dans des hot paths ou dans des opérations hautement répétitives.

Exemple

```java
public int countMatches(List<String> items, String target) {
    int count = 0;
    for (String s : items) {
        if (s.toLowerCase().equals(target.toLowerCase())) {
            count++;
        }
    }
    return count;
}
```

Interprétation :

- des appels répétés à `toLowerCase()` créent un travail non nécessaire
- le temps CPU augmente avec la taille de l'entrée
- calcul évitable dans les hot paths

Le problème n'est pas seulement le coût de la boucle en elle-même, mais la transformation répétée de valeurs qui pourraient être normalisées une seule fois au lieu de l'être à chaque comparaison.

Mécanisme

L'inefficacité CPU-bound gaspille de la capacité d'exécution.

Plus de temps CPU que nécessaire est consommé pour produire le même résultat.

Avec la croissance de la charge de travail :

- l'utilisation de la CPU augmente plus tôt
- le travail exécutable s'accumule plus tôt
- le throughput utile atteint plus tôt sa limite

Cela transforme un code inefficace en un goulot d'étranglement au niveau système lorsque le volume des requêtes augmente.

Impact sous charge

- augmentation de l'utilisation de la CPU
- réduction du throughput
- saturation de la CPU anticipée

Cela conduit à des retards de scheduling (→ 1.8.1 CPU behavior) et à une croissance non linéaire de la latence (→ 1.5.3 Non-linear degradation).

En termes pratiques, le système atteint sa propre limite CPU plus tôt que prévu, laissant moins de marge pour des bursts ou une croissance concurrente du trafic.

Symptômes observables

Les symptômes typiques incluent :

- utilisation élevée de la CPU sous charge modérée
- latence en augmentation avec l'augmentation du volume de requêtes
- throughput qui s'aplatit plus tôt que prévu
- temps CPU significatif passé dans des opérations répétées ou évitables

Ces symptômes apparaissent souvent avant la saturation totale de la CPU et peuvent initialement ressembler à un problème générique de scalabilité.

Implications pratiques

- optimiser les hot paths
- éviter le travail répété
- réduire la complexité algorithmique

Il est aussi important d'identifier quelles inefficacités comptent vraiment au niveau du système.

Une opération inefficace exécutée une fois peut être négligeable.

La même inefficacité exécutée des millions de fois devient un goulot d'étranglement.

Interprétation pratique

L'inefficacité CPU est l'une des raisons les plus communes pour lesquelles un système n'arrive pas à scaler malgré un hardware apparemment adéquat.

Le problème n'est pas le manque de CPU en termes absolus, mais la mauvaise utilisation de la CPU disponible.

L'optimisation est donc d'autant plus précieuse qu'elle augmente la quantité de travail utile accomplie par unité de temps CPU.

Idée clé

L'inefficacité CPU réduit la quantité de travail utile que le système peut accomplir avant d'atteindre la saturation.

1.9.2 Allocation excessive et churn mémoire

Définition

L'allocation excessive se vérifie lorsque le système crée un grand nombre d'objets à courte durée de vie, augmentant le churn mémoire et la pression sur le runtime.

C'est un problème commun dans les managed runtimes, où l'allocation est souvent peu coûteuse par opération, mais devient très coûteuse, en agrégat, lorsqu'elle est exécutée excessivement et sous charge.

Exemple

```
for (Order o : orders) {
    result.add(new ReportRow(o.getId(), o.getAmount(), o.getStatus()));
}
```

Interprétation :

- beaucoup d'objets sont créés par itération
- les objets ont une courte durée de vie
- le taux d'allocation augmente

Si ce pattern apparaît dans un code exécuté fréquemment, le volume total d'allocation peut devenir significatif même lorsque chaque objet individuel reste peu impactant.

Mécanisme

- un taux d'allocation élevé augmente le churn mémoire
- la garbage collection est exécutée plus fréquemment

(→ 1.7.2 Allocation and object lifecycle)
(→ 1.7.3 Garbage collection)

Le système souffre donc non seulement dans la phase de création des objets, mais pour les tracer, les éliminer et gérer, en général, les effets sur le runtime d'un turnover fréquent de la mémoire.

Impact sous charge

- augmentation de l'activité GC
- overhead CPU pour la gestion de la mémoire
- variabilité de la latence

Cela contribue à la pression sur la mémoire (→ 1.7.4 Memory pressure and performance).

Avec l'augmentation de la charge, l'overhead lié à l'allocation devient souvent plus visible à travers des pauses, du jitter et un élargissement des percentiles de latence.

Symptômes observables

Les symptômes typiques incluent :

- augmentation de la fréquence de la garbage collection
- pics périodiques de latence
- écart croissant entre latence moyenne et latence de queue
- utilisation modérée de la CPU avec des temps de réponse instables
- comportement de la mémoire qui se dégrade avec l'augmentation du throughput

Ces symptômes sont particulièrement communs dans les systèmes qui allouent fortement dans les chemins de traitement des requêtes.

Implications pratiques

- réduire la création non nécessaire d'objets
- réutiliser les objets lorsque c'est approprié
- analyser les patterns d'allocation

Il est aussi important de distinguer entre :

- allocation nécessaire
- allocation évitable

- allocation retenue qui aurait dû au contraire être temporaire

Cette distinction aide à déterminer si le problème est le churn, la rétention ou les deux.

Interprétation pratique

L'allocation excessive est souvent invisible en code review parce que le code reste simple et correct.

Son effet devient visible seulement à runtime, lorsque la création répétée d'objets change le comportement de la GC et la pression mémoire.

Un système peut donc apparaître logiquement efficient et malgré cela se comporter mal parce qu'il crée trop de trafic mémoire transitoire.

Idée clé

Le churn mémoire augmente l'overhead du runtime et introduit de la variabilité de la latence.

1.9.3 Contention et hot spots de synchronisation

Définition

La contention se vérifie lorsque plusieurs threads entrent en compétition pour la même ressource, forçant un accès sérialisé.

Un hot spot de synchronisation est une partie du système dans laquelle cette compétition devient concentrée et retarde répétitivement l'exécution.

Ces hot spots sont particulièrement problématiques parce qu'ils réduisent le parallélisme effectif exactement là où l'on s'attend à ce que la concurrence puisse aider.

Exemple

```
public class Counter {
    private int value = 0;

    public synchronized void increment() {
        value++;
    }
}
```

Interprétation :

- l'accès est sérialisé à travers la synchronisation
- un seul thread progresse à la fois
- le throughput est limité par la section critique

Le problème n'est pas que la synchronisation existe, mais qu'un chemin partagé et fréquemment accédé puisse devenir le point limitant pour l'ensemble du système.

Mécanisme

- les threads se bloquent en attendant le lock
- la contention augmente avec la concurrence

(→ 1.6 Concurrency and parallelism)

Lorsque plusieurs threads entrent en compétition pour la même section synchronisée :

- le temps d'attente croît

- le parallélisme effectif diminue
- plus de temps est dépensé dans la coordination que dans le progrès

Cela fait que le système se comporte comme si son niveau de concurrence était inférieur à ce que le nombre de threads suggère.

Impact sous charge

- augmentation du temps d'attente
- réduction du throughput
- augmentation de la latence

Cela conduit à des effets de mise en file d'attente (→ 1.5 System behavior under load).

Sous charge plus élevée, les hot spots de synchronisation deviennent souvent visibles sous la forme d'une croissance de la latence sans croissance proportionnelle de la CPU, parce que les threads sont en attente au lieu d'exécuter du travail.

Symptômes observables

Les symptômes typiques incluent :

- latence en augmentation avec utilisation modérée de la CPU
- beaucoup de threads bloqués ou en attente
- scalabilité réduite avec l'augmentation de la concurrence
- throughput limité par une petite section critique
- chemins de code avec usage intensif de locks qui apparaissent dans les hot paths d'exécution

Ces symptômes sont souvent trompeurs parce que le système peut apparaître seulement partiellement utilisé tout en étant déjà contraint.

Implications pratiques

- minimiser l'état mutable partagé
- réduire la taille de la section critique
- utiliser des patterns de concurrence plus scalables

Il est aussi important d'identifier si le goulot d'étranglement est causé par :

- scope du lock
- fréquence d'accès
- longues sections critiques
- synchronisation non nécessaire

Des causes différentes requièrent des solutions différentes.

Interprétation pratique

Les problèmes de contention sont souvent mal compris comme une lenteur générique.

En réalité, le problème central est la sérialisation : beaucoup de threads sont présents, mais seuls quelques-uns progressent dans le travail utile.

La performance engineering donc ne se préoccupe pas seulement d'ajouter de la concurrence, mais doit surtout s'assurer que la concurrence présente ne s'effondre pas en attente.

Idée clé

La contention convertit le travail parallèle en exécution sérialisée.

1.9.4 Goulots d'étranglement dus au blocking et à l'attente

Définition

Le blocking se vérifie lorsqu'un thread attend qu'une opération externe soit complétée, l'empêchant d'accomplir un travail utile.

Cela inclut l'attente de :

- I/O
- réponses réseau
- locks
- services externes
- autres événements coordonnés

Le blocking est souvent nécessaire, mais il devient un goulot d'étranglement lorsque trop de ressources d'exécution sont occupées à attendre au lieu de progresser.

Exemple

```
public String fetchData() throws Exception {
    Thread.sleep(50); // simulate blocking call
    return "data";
}
```

Interprétation :

- le thread est inactif pendant l'attente
- les ressources restent allouées
- la concurrence ne se traduit pas en throughput

Le thread existe, mais n'est pas en train de faire avancer du travail utile pendant la période de blocage.

Mécanisme

- les threads passent du temps à attendre au lieu d'exécuter
- les thread pools peuvent se saturer

(→ 1.6 Concurrency and parallelism)

Lorsque plusieurs threads se bloquent :

- moins de threads restent disponibles pour du nouveau travail
- la mise en file d'attente apparaît au niveau du modèle d'exécution
- la latence croît même si la CPU n'est pas pleinement utilisée

C'est la raison pour laquelle les goulots d'étranglement dus au blocking coexistent souvent avec une utilisation modérée de la CPU.

Impact sous charge

- augmentation de la latence
- réduction du throughput
- épuisement des threads

Cela amplifie la mise en file d'attente et la saturation (→ 1.5 System behavior under load).

Sous charge soutenue, le comportement bloquant crée souvent une boucle de feedback dans laquelle les requêtes en file attendent des threads qui, à leur tour, attendent des opérations lentes.

Symptômes observables

Les symptômes typiques incluent :

- beaucoup de threads dans des états d'attente ou bloqués
- files de requêtes en croissance
- CPU modérée avec throughput médiocre
- latence en augmentation pendant des opérations heavy en I/O ou heavy en dépendances
- thread pools qui apparaissent pleins sans travail productif correspondant

Ces symptômes sont particulièrement communs dans les services qui mélangent concurrence des requêtes et appels downstream synchrones.

Implications pratiques

- réduire les opérations bloquantes
- utiliser des patterns asynchrones ou non bloquants lorsque c'est approprié
- dimensionner avec attention les thread pools

Il est aussi utile de distinguer entre :

- blocking inévitable
- blocking évitable
- blocking placé dans des chemins d'exécution à haute fréquence

Cette distinction aide à identifier là où un redesign est nécessaire.

Interprétation pratique

Le blocking réduit la concurrence effective.

Un système peut avoir beaucoup de threads, mais si une grande partie d'entre eux est en attente, le système se comporte comme s'il avait beaucoup moins de capacité d'exécution.

C'est la raison pour laquelle les problèmes de blocking sont souvent des problèmes du modèle d'exécution avant de devenir des problèmes de pure ressource.

Idée clé

Le blocking réduit la concurrence effective et limite le throughput du système.

1.9.5 Accumulation de files d'attente et effets de saturation

Définition

L'accumulation de files d'attente se vérifie lorsque le travail entrant dépasse la capacité de traitement, causant l'attente des requêtes avant qu'elles soient traitées.

C'est l'un des problèmes de performance les plus communs et les plus importants, parce que le queueing transforme une surcharge peut-être modérée en une latence rapidement croissante.

Mécanisme

- le taux d'arrivée dépasse la capacité de service
- les files croissent dans le temps

Cela peut être décrit en utilisant Little's Law (→ 1.2.1 Little's Law (system-level concurrency)).

Pendant que la demande entrante continue et que le traitement reste limité, l'attente s'accumule et le temps de réponse commence à inclure un retard de file de plus en plus grand.

Impact sous charge

- le temps d'attente augmente
- le temps de réponse augmente
- la latence devient instable

Cela conduit à une dégradation non linéaire (→ 1.5.3 Non-linear degradation) et à des limites de throughput.

Une fois que la mise en file d'attente devient dominante, le système peut se détériorer très rapidement même si l'augmentation d'origine de la charge était relativement petite.

Symptômes observables

- longueurs de file croissantes
- temps de réponse en augmentation
- throughput stable ou en diminution

D'autres symptômes peuvent inclure :

- bursts d'erreurs de timeout
- élargissement de la latence p95/p99
- récupération retardée après une surcharge temporaire

Ces effets indiquent souvent que le système opère près ou au-delà de sa capacité effective.

Implications pratiques

- contrôler la concurrence
- augmenter la capacité de la ressource qui est le goulot d'étranglement
- réduire le taux d'arrivée si nécessaire

Il est aussi important de déterminer où la file est en train de se former :

- thread pool
- connection pool
- dispositif
- buffer réseau
- service downstream

La position de la file révèle souvent le vrai goulot d'étranglement.

Interprétation pratique

L'accumulation de files d'attente n'est pas seulement un détail opérationnel.

C'est souvent le mécanisme direct à travers lequel la surcharge devient visible pour les utilisateurs.

Un système peut encore fonctionner, mais une fois que le travail commence à attendre de manière systématique, la croissance de la latence devient inévitable.

Idée clé

Les files croissent lorsque la demande dépasse la capacité, déterminant la latence.

1.9.6 Amplification des dépendances et latence en cascade

Définition

L'amplification des dépendances se vérifie lorsque la latence dans un composant se propage et augmente la latence à travers le système.

Ce problème est particulièrement important dans les systèmes distribués, où une requête dépend souvent de plusieurs appels downstream avant de pouvoir se compléter.

Mécanisme

- les requêtes dépendent de plusieurs services downstream
- les retards s'accumulent à travers les appels
- des composants lents influencent l'ensemble du système

Même lorsque chaque retard individuel est petit, l'effet total peut devenir significatif une fois que plusieurs dépendances, retries ou chaînes d'appels sériels sont impliquées.

Exemple

```
public Response process() {
    Data a = serviceA.call();
    Data b = serviceB.call();
    return combine(a, b);
}
```

Interprétation :

- la latence totale dépend de plusieurs dépendances
- la dépendance la plus lente domine le temps de réponse

Dans les systèmes réels, cet effet devient plus fort lorsque les requêtes dépendent de nombreux services, de bases de données distantes ou d'opérations synchrones enchaînées.

Impact sous charge

- amplification de la latence à travers les services
- augmentation de la variabilité
- dégradation de la latence de queue

(→ 1.5.5 Tail latency amplification)

Sous charge, l'amplification des dépendances devient souvent plus sévère parce que des systèmes downstream lents retiennent des threads, des requêtes et des files upstream pendant des périodes plus longues.

Symptômes observables

Les symptômes typiques incluent :

- augmentations soudaines de latence sans saturation locale de la CPU
- dégradation du comportement p95/p99 causée par la variabilité downstream
- chaînes de requêtes qui deviennent plus lentes pendant qu'une dépendance ralentit
- instabilité qui se diffuse d'un service à un autre
- retries et timeouts qui augmentent la pression à travers le système

Ces symptômes sont souvent difficiles à interpréter sans corréler le comportement à travers plusieurs composants.

Implications pratiques

- minimiser le nombre de dépendances synchrones
- utiliser des timeouts et des stratégies de fallback
- isoler les composants lents

Il est aussi utile d'identifier :

- quelle dépendance contribue le plus au retard end-to-end
- si les appels sont sériels ou parallèles
- si les retries aggravent le problème
- si les composants lents déclenchent une mise en file d'attente upstream

Cela transforme un vague problème de "lenteur distribuée" en un comportement système diagnostiable.

Interprétation pratique

La latence d'un système n'est pas déterminée seulement par son "propre code".

Elle est souvent déterminée par la dépendance la plus lente dans le chemin de la requête.

Plus un système a de dépendances, plus il est probable que la variabilité à un endroit devienne visible partout.

Idée clé

La latence du système est souvent déterminée par la dépendance la plus lente.

◀ 01-08-resource-level-performance | ▲ Index | 01-10-diagnostics-and-analysis ▶

1.10 – Diagnostic et analyse

Ce chapitre s'intéresse à la manière dont les problématiques de performance peuvent être investiguées, interprétées et validées.

On se concentre ici sur les processus utilisés pour passer de l'observation du système à une évaluation défendable de la performance de celui-ci.

Le diagnostic en effet n'est pas seulement une pratique de collecte des données.

C'est la discipline qui se préoccupe d'interpréter correctement ces données et de relier les symptômes aux mécanismes de fonctionnement sous-jacents.

Table des matières

1.10.1 Observabilité et signaux

Définition

Le diagnostic part évidemment de signaux observables.

Ces signaux fournissent une visibilité souvent indirecte sur le comportement interne du système sous charge.

Ils n'exposent pas directement les mécanismes de fonctionnement, mais en reflètent plutôt les effets.

Pour cette raison l'observabilité est essentielle dans la performance engineering : les problèmes internes sont rarement visibles directement, mais ils laissent souvent des traces mesurables en ce qui concerne la latence, le throughput, le comportement des ressources et le queueing.

Signaux fondamentaux

Les signaux primaires sont :

- latence (p50, p95, p99)
- throughput
- taux d'erreur
- utilisation des ressources (CPU, mémoire, I/O, réseau)
- longueurs des files

(→ 1.2 Core metrics and formulas)
(→ 1.8 Resource-level performance)

Chaque signal capture une dimension différente du comportement du système.

Seul un examen combiné de ceux-ci fournit une vue significative du système.

- La latence montre l'impact visible pour l'utilisateur.
- Le throughput montre le taux de travail productif.
- Le taux des erreurs indique le comportement en cas de défaillance.
- Les indices des ressources montrent où la capacité est consommée.
- Les files montrent où le travail s'accumule.

Caractéristiques des signaux

Les signaux doivent être :

- **précis** → refléter le comportement réel
- **granulaires** → exposer la distribution (ex. percentiles, pas seulement moyennes)
- **corrélés dans le temps** → alignés à travers tous les composants

Sans ces propriétés, l'interprétation devient peu fiable, trompeuse ou même erronée.

Une métrique mal placée, pas bien configurée ou même déconnectée de l'intervalle de temps pertinent peut cacher précisément ce mécanisme de fonctionnement qu'elle entend au contraire révéler.

Qualité du signal et interprétation

La présence de signaux n'est toutefois pas à elle seule suffisante.

Les signaux doivent aussi être :

- pertinents par rapport aux questions que l'on se pose
- observés par rapport au niveau approprié (système, service, ressource, dépendance)
- interprétés dans le contexte

Par exemple :

- l'utilisation CPU sans information sur la run queue peut cacher de la pression de scheduling
- la latence moyenne sans analyse des percentiles peut cacher de l'instabilité en queue
- l'utilisation de la mémoire sans comportement de la GC peut cacher de la pression du runtime

La valeur diagnostique d'une métrique dépend non seulement de son existence, mais de la manière dont elle est corrélée avec le reste des évidences.

Implications pratiques

Un diagnostic efficace nécessite de :

- observer de manière globale les signaux
- corréler ces signaux dans le temps
- éviter le raisonnement basé sur des métriques uniques

Observer une métrique en dehors d'un contexte est souvent trompeur en ce qui concerne la compréhension de la mécanique sous-jacente.

C'est l'une des principales raisons pour lesquelles des explications simplistes sont dangereuses dans l'analyse des performances.

Un seul nombre peut décrire un symptôme, mais explique rarement le comportement global du système en question.

Interprétation pratique

L'observabilité est la matière première du diagnostic.

Sans signaux, il n'existe pas d'analyse fiable.
Avec des signaux de mauvaise qualité, l'analyse sera peu fiable.
Avec des signaux bien structurés, l'analyse devient vérifiable et répétable.

Le diagnostic commence donc non avec l'optimisation, mais avec l'analyse de ce qui est observé.

Idée clé

Le diagnostic dépend à la fois de la disponibilité et de la correcte interprétation des signaux observables.

1.10.2 Symptôme vs cause

Définition

Un symptôme est un effet observable.

Une cause est le mécanisme sous-jacent qui produit cet effet.

Cette distinction est fondamentale parce que la majorité des problèmes de performance est découverte à travers des symptômes, non à travers une manifestation directe de la cause à la racine du problème.

Distinction

Symptômes typiques :

- latence élevée
- utilisation importante de la CPU
- augmentation du taux d'erreur
- garbage collection fréquente

Ces éléments décrivent *ce qui se passe*, non *pourquoi cela se passe*.

Un système peut montrer le même symptôme pour des raisons très différentes, et la même cause peut produire des symptômes différents selon la charge, le timing et l'architecture.

Exemple

- une utilisation élevée de la CPU peut résulter de :
    - calcul inefficace
    - retries excessifs
    - pression de mémoire
    - contention
- une latence élevée peut résulter de :
    - accumulation de files
    - retards d'I/O
    - synchronisation

(→ 1.9 Common performance problems)

Pour ces raisons les symptômes doivent être traités comme des points d'accès à l'investigation, non comme des explications.

Implication diagnostique

Le même symptôme peut être produit par des causes différentes.

Sans identifier le mécanisme sous-jacent, les actions correctives peuvent prendre pour cible la mauvaise partie du système.

Par exemple :

- réduire l'utilisation de la CPU peut ne pas réduire la latence si la cause racine est le queueing I/O
- faire du tuning de la GC peut ne pas aider si le taux d'allocation d'objets reste inchangé

Un fix techniquement plausible peut donc avoir peu d'effet s'il ne traite qu'une seule conséquence visible.

Pourquoi la confusion se produit

Symptômes et causes sont souvent confondus parce que les symptômes sont relativement faciles à observer.

Les métriques, les dashboards et les systèmes de monitoring montrent habituellement :

- des valeurs élevées
- ce qui est lent
- ce qui est en train d'échouer

Ils n'expliquent pas automatiquement :

- pourquoi les valeurs sont élevées
- pourquoi c'est lent
- pourquoi cela est en train d'échouer

Cet écart entre visibilité et explication est exactement ce que le diagnostic doit combler.

Interprétation pratique

Un bon processus diagnostique traite chaque symptôme comme un indice, non comme une conclusion.

L'objectif est de passer de :

- "cette métrique est anormale"

à :

- "ce mécanisme est en train de produire le comportement anormal"

Ce déplacement est ce qui distingue un raisonnement efficace sur les performances d'un monitoring superficiel.

Idée clé

Le comportement observé n'est pas la cause.

Le diagnostic requiert de mapper les symptômes aux mécanismes sous-jacents qui les génèrent.

1.10.3 Corrélation et causalité

Définition

La corrélation est la variation simultanée de deux signaux.

La causalité est une relation directe dans laquelle un facteur en produit un autre.

Cette distinction est essentielle dans le diagnostic parce que beaucoup de métriques évoluent ensemble sous charge, mais elles ne sont pas toutes causalement reliées dans la même direction.

Erreur commune

Deux métriques changent ensemble :

- la CPU augmente
- la latence augmente

Cela n'implique pas que la CPU soit la cause de la latence.

La corrélation peut indiquer :

- une cause sous-jacente commune
- une dépendance indirecte
- une chaîne causale dans la direction opposée
- ou une simple coïncidence dans la même fenêtre temporelle

Exemple

Interprétations possibles :

- saturation CPU → retards de scheduling → latence
- retards d'I/O → plus de threads concurrents → plus grande utilisation de la CPU
- contention → retries → CPU et latence augmentent toutes deux

(→ 1.5 System behavior under load)
(→ 1.8 Resource-level performance)

Dans les trois cas, CPU et latence évoluent ensemble, mais le mécanisme sous-jacent est différent.

Implication diagnostique

La corrélation est un point de départ, non une conclusion.

Plusieurs mécanismes peuvent produire les mêmes signaux corrélés.

Seul un modèle causal explique comment l'un conduit à l'autre.

Pour cette raison, le raisonnement diagnostique doit aller au-delà de "ces deux métriques ont évolué au même moment".

Il doit expliquer :

- laquelle a changé en premier
- quel mécanisme les relie
- pourquoi la séquence observée est cohérente avec le comportement du système

---

Approche pratique

Pour établir la causalité :

- identifier la séquence des événements
- vérifier la cohérence avec le comportement connu du système
- valider à travers l'observation ou un changement contrôlé

Cela peut inclure :

- comparer les états avant/après
- observer si une métrique précède constamment une autre
- changer une condition et vérifier la réponse attendue

La causalité devient plus forte lorsque le système se comporte comme le prévoit le mécanisme proposé.

Limites de l'analyse superficielle

Un dashboard peut montrer la corrélation très clairement mais ne peut pas, à lui seul, prouver la causalité.

Pour cette raison le diagnostic requiert du raisonnement et pas seulement de la "visualisation".

Un performance engineer doit se demander :

- Cette métrique est-elle le driver, la conséquence ou une conséquence supplémentaire du même événement ?
- La timeline supporte-t-elle l'explication proposée ?
- L'explication reste-t-elle cohérente à travers des observations répétées ?

Sans ces questions, la corrélation peut facilement conduire à des conclusions incorrectes.

Interprétation pratique

Un bon diagnostic traite la corrélation comme un générateur d'hypothèses.

Cela aide à identifier où regarder, mais n'élimine pas la nécessité de raisonner sur les mécanismes sous-jacents.

Cela est particulièrement important dans les systèmes complexes où plusieurs goulots d'étranglement interagissent et où les symptômes se propagent à travers les composants.

Idée clé

Ne pas inférer la causalité à partir de la corrélation.

Le diagnostic requiert d'identifier le mécanisme qui relie les signaux.

1.10.4 Construire une hypothèse

Définition

Une hypothèse est une explication proposée qui relie des signaux observés à un mécanisme du système.

Elle fournit une manière structurée de passer de l'observation à l'explication.

Sans hypothèse, l'analyse reste descriptive plutôt que diagnostique.

Processus

Une hypothèse est construite :

1. en observant les signaux
2. en identifiant des patterns cohérents
3. en les mappant sur des mécanismes connus

(→ 1.2 Core metrics and formulas)
(→ 1.5 System behavior under load)

Ce processus transforme des données brutes en une explication testable.

Il relie :

- mesures
- comportement du système
- raisonnement causal

Exemple

Observé :

- la latence augmente
- la longueur de la file augmente
- la CPU s'approche de la saturation

Hypothèse :

- augmentation du taux de travail entrant → accumulation de file → temps d'attente plus long → saturation CPU

Cela relie des signaux observables à un mécanisme de mise en file d'attente.

Cela fournit aussi une direction à l'investigation : vérifier si l'augmentation de la latence est causée principalement par l'attente plutôt que par un temps de service plus lent.

Exigences

Une hypothèse valide doit être :

- cohérente avec les données observées
- fondée sur le comportement du système
- testable à travers mesure ou changement

Une hypothèse qui ne peut pas être testée peut être plausible, mais elle n'est pas encore utile pour le diagnostic.

Une hypothèse qui contredit l'évidence observée devrait être rejetée même si elle paraît intuitive.

Implication diagnostique

Une hypothèse guide l'investigation.

Sans elle, l'analyse devient réactive et non structurée.

Au lieu de passer directement du symptôme au fix, le processus diagnostique devrait passer de :

- symptôme
- hypothèse sur un mécanisme candidat
- validation

Cette structure réduit le guesswork et rend les conclusions diagnostiques plus robustes.

Sources des hypothèses

Les hypothèses émergent habituellement de :

- combinaisons de signaux observés
- patterns de performance connus
- comportement précédent du système
- connaissance architecturale
- scénarios d'erreur répétés

Par exemple :

- latence croissante + files en croissance suggère souvent de la mise en file d'attente
- CPU modérée + threads bloqués peut suggérer de la contention ou de l'attente I/O
- fréquence GC croissante + pics de latence peut suggérer de la pression de mémoire

Ces associations ne prouvent pas l'explication, mais fournissent un point de départ discipliné.

Interprétation pratique

Une bonne hypothèse est suffisamment spécifique pour être testée et suffisamment générale pour expliquer le comportement observé.

Elle ne devrait pas être :

- vague ("le système est lent")
- circulaire ("la latence est élevée parce que les requêtes sont lentes")
- purement descriptive

Elle devrait exprimer un mécanisme.

Par exemple :

- "La saturation du thread pool est en train d'augmenter le temps de file, ce qui fait monter la latence p95."

Ce type d'affirmation peut être validé.

Idée clé

Le diagnostic procède à travers des hypothèses explicites et testables, non à travers des suppositions non reliées.

1.10.5 Réduire le goulot d'étranglement

Définition

Le diagnostic vise à identifier la ressource ou le mécanisme qui limite la performance du système.

Ce facteur limitant détermine le comportement global du système sous charge.

Tant qu'il n'est pas identifié, les efforts d'optimisation restent incertains et souvent inefficaces.

Approche

L'analyse se concentre sur :

- comportement de la CPU
- latence I/O
- retards réseau

- pression de mémoire

(→ 1.8 Resource-level performance)
(→ 1.7 Runtime and memory model)

Ces dimensions sont examinées parce que la majorité des limites de performance, à la fin, se manifeste à travers une ou plusieurs d'entre elles.

Cependant, le goulot d'étranglement dominant, à un moment donné, est d'habitude donné par une seule contrainte primaire plutôt que par toutes les contraintes à égalité.

Méthode

- isoler une dimension à la fois
- comparer les signaux à travers les ressources
- identifier la contrainte dominante

Cela réduit la complexité en se concentrant sur le facteur le plus impactant.

L'objectif n'est pas d'expliquer chaque métrique, mais de trouver le mécanisme qui, à ce moment-là, gouverne le comportement du système.

Exemple

Si :

- la CPU est basse
- la latence I/O est élevée
- les files sont en train de croître

Alors :

- l'I/O est probablement le facteur limitant

Le système n'est pas CPU-bound, même si la CPU est active.

Ce type de réduction est essentiel parce que plusieurs ressources sont souvent impliquées, mais une seule d'entre elles est habituellement dominante.

Implication diagnostique

La performance est typiquement limitée, à un moment donné, par un seul goulot d'étranglement dominant.

Optimiser des ressources non limitantes produit peu ou pas d'amélioration.

C'est l'un des principes les plus importants dans le diagnostic :

- mesurer de manière large
- conclure de manière spécifique

Un ensemble large de signaux est requis pour éviter de perdre des évidences importantes.

Une conclusion spécifique est requise pour orienter l'action sur la contrainte réelle.

Pourquoi les goulots d'étranglement sont difficiles à identifier

Les goulots d'étranglement sont souvent obscurcis par des effets secondaires.

Par exemple :

- un I/O lent peut augmenter le nombre de threads
- l'augmentation du nombre de threads peut augmenter l'overhead de scheduling de la CPU
- l'augmentation de l'attente peut gonfler la rétention de mémoire

- les retries peuvent amplifier la demande sur plusieurs composants en même temps

Par conséquent, l'effet visible peut ne pas apparaître au point exact du problème d'origine.

Pour cette raison l'isolement du goulot d'étranglement requiert de la corrélation à travers les layers plutôt qu'une interprétation isolée d'une métrique unique.

Interprétation pratique

Le but du diagnostic n'est pas seulement de dire que le système est sous pression.

C'est d'identifier :

- où la pression devient limitante
- quel mécanisme produit la limite
- pourquoi cette contrainte est actuellement dominante

Seulement alors l'optimisation devient significative.

Idée clé

Un diagnostic efficace réduit le système à son facteur limitant.

1.10.6 Analyse itérative et validation

Définition

Le diagnostic est un processus itératif de test et d'affinement des hypothèses.

Il évolue à travers des observations et des validations successives.

Cela est nécessaire parce que les explications initiales sont souvent incomplètes, partiellement correctes ou valides seulement pour un layer du système.

Processus

1. observer les signaux
2. construire une hypothèse
3. tester à travers des changements ou des mesures
4. valider ou rejeter

Chaque passage produit un affinement dans la compréhension du système.

Cette boucle est répétée jusqu'à ce que l'explication proposée soit cohérente avec le comportement observé et supportée par l'évidence.

Exemple

```
ExecutorService pool = Executors.newFixedThreadPool(10);

for (int i = 0; i < 1000; i++) {
    pool.submit(() -> {
        Thread.sleep(100);
        return null;
    });
}
```

Interprétation :

- le thread pool fixe limite l'exécution parallèle

- les tâches s’accumulent
- la mise en file d’attente augmente la latence

Cette hypothèse peut être testée :

- en augmentant la taille du pool
- en réduisant le temps de blocking

Si la latence diminue et que l’accumulation de files se réduit, l’hypothèse gagne en évidence.

Si le comportement ne change pas comme prévu, l’explication doit être révisée.

Validation

Une hypothèse est validée si :

- les changements produisent les effets attendus
- les signaux évoluent de manière cohérente avec le mécanisme proposé

Dans le cas contraire, l’hypothèse doit être révisée.

La validation dépend donc de la cohérence entre :

- changement observé
- changement attendu
- explication causale proposée

Un fix qui change une métrique sans améliorer le comportement du système peut indiquer que le mauvais mécanisme a été visé.

Implications pratiques

- éviter les conclusions en une seule étape
- itérer systématiquement
- valider les suppositions avec des données observables

Un bon diagnostic est rarement instantané.

Il devient fiable à travers une comparaison répétée entre :

- ce qui est observé
- ce qui est attendu
- ce qui change réellement après l’intervention

Cette discipline itérative est ce qui transforme le troubleshooting en engineering.

Pourquoi l’itération compte

Les systèmes complexes exposent rarement une explication complète dans une seule observation.

Il est commun de découvrir que :

- un goulot d’étranglement initial n’était qu’un effet secondaire
- supprimer une contrainte en expose une autre
- une amélioration locale déplace ailleurs le facteur limitant
- le système se comporte différemment sous des charges de travail différentes

L’itération n’est donc pas un signe d’incertitude.

C’est la méthode normale pour arriver à une explication cohérente.

Interprétation pratique

Le diagnostic est une boucle parce que la compréhension du système se construit progressivement.

L'objectif n'est pas de deviner correctement à la première tentative.

L'objectif est de passer de l'évidence à l'explication à travers un raisonnement contrôlé et une vérification.

C'est ce qui rend l'analyse des performances répétable et défendable.

Idée clé

Le diagnostic est une boucle.

La compréhension émerge à travers itération, vérification et affinement.

◀ 01-09-common-performance-problems | ▲ Index | 01-11-practical-checklists ▶

1.11 – Checklists pratiques

Ce chapitre fournit des checklists pratiques pour préparer, exécuter et analyser des tests de performance.

À la différence des chapitres précédents, qui expliquent des concepts et des mécanismes, ce chapitre se concentre sur la discipline opérationnelle.

L'objectif est de réduire les erreurs évitables et d'assurer que les tests de performance produisent des résultats interprétables, fiables et utiles.

Table des matières

1.11.1 Avant d'exécuter un test

Objectifs

Définir clairement ce que le test entend valider.

Des objectifs typiques incluent :

- cibles de latence
- objectifs de throughput
- limites de capacité

Un test sans objectif clair peut quand même générer des données, mais ces données seront difficiles à évaluer.

La première question devrait toujours être :

- qu'est-ce que ce test devrait prouver, valider ou révéler ?

Définition de la charge de travail

Définir la charge de travail avec précision :

- taux de requêtes ou concurrence

- mix de requêtes
- durée

(→ 1.4 Types of performance tests)

La charge de travail doit être suffisamment spécifique pour être reproductible et suffisamment réaliste pour être significative.

Une charge de travail vague ou artificielle peut produire des résultats techniquement corrects mais opérationnellement non pertinents.

Cohérence de l'environnement

S'assurer que :

- l'environnement de test soit stable
- la configuration corresponde aux hypothèses de production
- les dépendances externes soient contrôlées

Si l'environnement change pendant le testing, l'interprétation devient incertaine.

Les résultats de performance sont comparables seulement si les conditions d'exécution restent suffisamment cohérentes.

Cela est particulièrement important lorsque l'on évalue :

- des changements de configuration
- des changements de code
- des changements infrastructurels

Setup des métriques

Vérifier que toutes les métriques requises soient disponibles :

- percentiles de latence
- throughput
- utilisation des ressources
- taux d'erreur

(→ 1.2 Core metrics and formulas)

Il est aussi utile de s'assurer que des signaux de support soient disponibles lorsqu'ils sont pertinents, comme :

- longueurs des files
- timing des dépendances
- activité GC
- états des threads ou des pools

Le test ne devrait pas commencer avant que la visibilité soit en place.

Contrôles de préparation

Avant d'exécuter le test, confirmer que :

- le système cible soit dans l'état attendu
- le monitoring soit actif
- le générateur de charge de travail soit configuré correctement
- la durée du test soit appropriée pour l'objectif choisi
- les critères de succès et d'échec soient connus à l'avance

Cela évite un problème commun dans le performance testing : exécuter un test techniquement valide qui ensuite ne peut pas être interprété avec certitude.

Interprétation pratique

La préparation fait partie du test.

La majorité des résultats peu fiables n'est pas causée par un comportement complexe du système, mais par une mauvaise préparation du test :

- objectifs peu clairs
- charge de travail non réaliste
- environnement incohérent
- métriques incomplètes

Un test bien préparé rend le diagnostic successif beaucoup plus facile.

Idée clé

Un test est significatif seulement si les objectifs, la charge de travail et les mesures sont clairement définis.

1.11.2 Pendant l'exécution du test

Monitoring

Observer le comportement du système en temps réel :

- évolution de la latence
- stabilité du throughput
- utilisation des ressources

Le monitoring pendant l'exécution est important parce que certains problèmes sont visibles seulement pendant que le test est en cours d'exécution, spécialement :

- saturation soudaine
- mise en file d'attente inattendue
- récupération instable
- défaillances des dépendances

Attendre la fin du test peut cacher des comportements importants dépendants du temps.

Contrôles de cohérence

S'assurer que :

- la charge de travail soit appliquée comme prévu
- aucune perturbation externe n'influence le test

Cela inclut vérifier que :

- le taux de requêtes prévu soit effectivement généré
- le mix d'opérations reste cohérent
- aucune activité non corrélée ne soit en train de distordre les résultats
- les échecs soient causés par les conditions de test plutôt que par du bruit externe

Une divergence entre charge de travail prévue et charge de travail réelle peut invalider l'interprétation entière.

Signaux précoces

Observer :

- augmentation rapide de la latence
- erreurs inattendues
- saturation des ressources

(→ 1.8 Resource-level performance)

Ceux-ci sont souvent les premiers signaux que le système est en train de s'approcher d'une limite ou que la charge de travail est en train d'exposer un goulot d'étranglement non anticipé.

L'identification précoce est importante parce qu'elle permet à l'opérateur du test de :

- capturer des évidences pertinentes
- préserver un contexte utile
- éviter de perdre la partie la plus informative de l'exécution

---

Observations à runtime

Pendant l'exécution, il est utile d'observer non seulement des valeurs absolues, mais aussi le changement dans le temps.

Exemples :

- latence en augmentation tandis que le throughput reste stable
- longueurs des files en croissance avant la saturation de la CPU
- erreurs qui apparaissent seulement après un seuil spécifique
- dégradation de p95/p99 avant que la moyenne change significativement

Ces patterns révèlent souvent plus que des snapshots isolés.

Ils aident à distinguer entre :

- instabilité transiente
- surcharge stable
- dégradation lente
- effondrement soudain

Discipline d'intervention

Pendant un test, éviter de changer des paramètres à moins que le changement ne fasse partie du plan de test.

Une intervention non planifiée rend les résultats plus difficiles à interpréter parce qu'elle mélange des causes multiples dans la même fenêtre d'observation.

Si l'intervention devient nécessaire, elle devrait être :

- documentée
- marquée temporellement
- explicitement reliée au comportement observé

Cela préserve la valeur diagnostique de l'exécution.

Interprétation pratique

L'exécution est la phase où la préparation théorique rencontre le comportement réel du système.

Un test bien conçu peut quand même devenir trompeur si l'opérateur ne confirme pas que :

- la charge de travail soit correcte

- l'environnement reste stable
- le système soit en train de se comporter comme prévu ou, chose importante, de manière inattendue comme le test entendait révéler

---

Idée clé

L'exécution n'est pas passive.

Une observation continue est requise pour détecter précocement les anomalies.

1.11.3 Après l'analyse du test

Révision des données

Analyser les données recueillies :

- distribution de la latence
- tendances de throughput
- utilisation des ressources

La révision des données devrait se concentrer non seulement sur les valeurs moyennes, mais aussi sur la forme du comportement dans le temps.

Par exemple :

- quand la dégradation a commencé
- si le throughput a scalé comme prévu
- si la latence en queue s'est élargie avant que les échecs apparaissent

Cela rend l'analyse plus diagnostique et moins descriptive.

Corrélation

Mettre en relation les signaux :

- latence vs CPU
- latence vs I/O
- erreurs vs charge

(→ 1.10 Diagnostics and analysis)

La corrélation aide à identifier quelle ressource ou quel mécanisme soit le plus probablement associé à la dégradation observée.

Toutefois, la corrélation devrait être traitée comme un point de départ analytique, non comme une conclusion finale.

Interprétation

Identifier :

- goulots d'étranglement
- limites de scalabilité
- patterns anormaux

L'interprétation devrait répondre à des questions comme :

- qu'est-ce qui a changé en premier ?
- qu'est-ce qui s'est dégradé après ?
- quelle contrainte est devenue dominante ?

- la dégradation a-t-elle été graduelle, brusque ou dépendante du temps ?

C'est le point où les mesures brutes deviennent compréhension du système.

Reporting

Résumer :

- comportement observé
- problèmes identifiés
- recommandations

Un rapport utile fait plus qu'énumérer des nombres.

Il devrait expliquer :

- ce que le système était censé faire
- ce qu'il a effectivement fait
- où il s'est écarté des attentes
- quelle évidence supporte la conclusion

Cela rend les résultats utilisables pour engineering, operations et tests futurs.

Orientation vers les étapes suivantes

Après l'analyse, définir ce qui devrait arriver ensuite.

Cela peut inclure :

- réexécuter le même test après modifications
- affiner le réalisme de la charge de travail
- recueillir un diagnostic plus profond
- isoler un goulot d'étranglement suspecté
- étendre vers des tests de stress, soak ou capacité

Sans une décision sur les étapes suivantes, l'analyse reste informative mais non utile opérationnellement.

Interprétation pratique

L'analyse post-test est le point où la performance engineering devient prise de décision.

Le but n'est pas seulement de déclarer qu'une métrique a changé, mais d'expliquer :

- pourquoi le changement est important
- ce qu'il implique sur le système
- ce qui devrait être fait ensuite

Idée clé

L'analyse transforme des données brutes en compréhension actionnable.

1.11.4 Erreurs communes

Mal interpréter les moyennes

- les moyennes cachent la latence en queue
- les percentiles fournissent une vue plus claire

(→ 1.2.7 Percentiles)

Un système peut apparaître sain en moyenne tout en produisant des performances inacceptables pour une fraction significative des requêtes.

C'est l'une des erreurs les plus communes dans l'interprétation des tests.

Ignorer le réalisme de la charge de travail

- des charges de travail non réalistes produisent des résultats trompeurs
- les patterns de production doivent être approximés

Une charge de travail synthétique peut être plus facile à générer, mais si elle ne reflète pas le réel mix de requêtes, la concurrence et le comportement des dépendances, les conclusions peuvent ne pas se transférer aux conditions de production.

Le réalisme ne requiert pas une reproduction parfaite, mais requiert une approximation crédible.

Confondre symptôme et cause

- une CPU élevée n'est pas toujours le problème à la racine
- la latence doit être analysée dans le contexte

(→ 1.10 Diagnostics and analysis)

Cette erreur conduit souvent à une optimisation inefficace.

Le symptôme visible peut être seulement la conséquence d'un mécanisme plus profond comme la mise en file d'attente, le blocking ou le ralentissement d'une dépendance.

Négliger les goulots d'étranglement

- optimiser des ressources non limitantes a peu d'effet
- le focus doit rester sur la contrainte dominante

(→ 1.8 Resource-level performance)

Ceci est une source fréquente d'effort gaspillé.

Un système peut contenir de nombreuses imperfections, mais seules certaines d'entre elles comptent au point opérationnel courant.

Exécuter des tests sans critères d'acceptation

Un test est difficile à interpréter s'il n'existe pas de définition préalable de comportement acceptable.

Sans seuils explicites, il devient peu clair si le résultat signifie :

- succès
- échec
- dégradation
- risque acceptable

Les nombres de performance sont utiles seulement lorsqu'ils sont comparés à des attentes définies.

Traiter un seul test comme définitif

Une seule exécution de test capture rarement le comportement complet d'un système.

Des exécutions différentes peuvent exposer :

- effets de warm-up
- variabilité des dépendances

- drift à long terme
- comportement de seuil sous des profils de charge différents

Une analyse de performance fiable requiert habituellement comparaison, répétition et validation.

Ignorer la dimension temporelle

Certains problèmes n'apparaissent pas immédiatement.

Un test court peut manquer :

- croissance lente de la mémoire
- accumulation retardée des files
- dégradation graduelle des dépendances
- instabilité du runtime dans le temps

Pour cette raison la durée du test doit correspondre au type de comportement que l'on est en train d'évaluer.

Interprétation pratique

La majorité des erreurs dans le performance testing n'est pas causée par de mauvais outils.

Elle est causée par :

- hypothèses faibles
- visibilité incomplète
- mauvaise interprétation
- manque de discipline méthodologique

Éviter ces erreurs est souvent plus précieux qu'ajouter davantage de détail de mesure.

Idée clé

Des hypothèses incorrectes conduisent à des conclusions incorrectes.

Éviter les erreurs communes est essentiel pour une analyse de performance fiable.

◀ 01-10-diagnostics-and-analysis | ▲ Index | ▶