E-fólio B | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Sistemas Distribuídos

CÓDIGO: 21108

DOCENTE: Nelson Russo

A preencher pelo estudante

NOME: Luís Carlos Crispim Pereira

N.º DE ESTUDANTE: 2300163

CURSO: Licenciatura Engenharia de Informática

DATA DE ENTREGA: 20/05/25

TRABALHO / RESOLUÇÃO:

Questão 3.1:

Avalie o impacto da utilização de middleware orientado a mensagens (como o Apache Kafka ou RabbitMQ) na construção de Sistemas Distribuídos escaláveis e resilientes. Que desafios surgem na garantia da ordem e entrega das mensagens?

Resposta 3.1:

A adoção de *middleware* orientado a mensagens (MOM) tornou-se um elemento central na construção de Sistemas Distribuídos modernos. Plataformas como Apache Kafka e RabbitMQ fornecem mecanismos de comunicação assíncrona e desacoplada entre componentes. Esta abordagem promove a escalabilidade horizontal e a resiliência a falhas, características fundamentais em arquiteturas de larga escala (Coulouris et al., 2011).

Estes sistemas baseiam-se no modelo *publish-subscribe* ou em filas de mensagens, em que produtores enviam dados para tópicos ou filas e os consumidores processam-nos de forma independente. Esta separação temporal e lógica entre produtores e consumidores permite que os sistemas evoluam e escalem autonomamente. Além disso, reduz o acoplamento entre componentes e facilita a tolerância a falhas (Hohpe & Woolf, 2003).

O Apache Kafka, por exemplo, é amplamente utilizado em plataformas como LinkedIn e Netflix devido à sua elevada taxa de *throughput* e ao armazenamento persistente de *logs* distribuídos. Este *middleware* assegura durabilidade, replicação entre *brokers* e tolerância a falhas de nós individuais. Por sua vez, o RabbitMQ é mais orientado a aplicações empresariais e destaca-se pela flexibilidade no encaminhamento de mensagens e pelo suporte a múltiplos protocolos, como AMQP e MQTT. Este último é ideal em sistemas que exigem compatibilidade heterogénea (Coulouris et al., 2011).

Apesar das vantagens, surgem desafios importantes ao nível da entrega e da ordenação das mensagens. Um dos principais problemas é a garantia da ordem. No Kafka, a ordem só é garantida dentro de uma partição específica. Se forem utilizadas múltiplas partições para paralelismo, a ordem global perde-se. Este comportamento exige uma arquitetura consciente do particionamento e, em muitos casos, lógica adicional na aplicação para reconstruir sequências (Boykin & Ritchie, 2020).

Outro desafio é a entrega garantida, frequentemente classificada em três níveis: *at-most-once*, *at-least-once* e *exactly-once*. Por padrão, o Kafka opera com semântica *at-least-once*, o que pode originar duplicação. Já o suporte a *exactly-once* implica compromissos em termos de desempenho, exigindo coordenação entre produtor, *broker* e consumidor (Kreps, 2014). O RabbitMQ, com mecanismos de confirmação e filas persistentes, tenta mitigar perdas, mas continua vulnerável em falhas abruptas.

A resiliência global depende também da replicação e do *failover* automático. Empresas como Uber e Spotify operam *clusters* Kafka com múltiplos *brokers* e coordenação via Zookeeper. Contudo, estas infraestruturas requerem configuração cuidada e monitorização constante, com recurso a ferramentas como Prometheus e Grafana.

Em síntese, *middleware* como Kafka e RabbitMQ são pilares fundamentais para sistemas escaláveis e tolerantes a falhas. No entanto, introduzem complexidades na ordenação, duplicação e entrega fiável de mensagens. O sucesso da sua utilização depende de decisões arquitetónicas conscientes sobre partições, semântica de entrega e mecanismos de recuperação.

Referências Bibliográficas 3.1:

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.

Hohpe, G., & Woolf, B. (2003). Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley.

Boykin, P., & Ritchie, J. (2020). Kafka: The Definitive Guide (2nd ed.). O'Reilly Media.

Kreps, J. (2014). I Heart Logs: Event Data, Stream Processing, and Data Integration. O'Reilly Media.

Questão 3.2:

Considere a integração de serviços heterogéneos utilizando middleware baseado em Web Services. Explique como os padrões SOAP e REST abordam a interoperabilidade e segurança entre diferentes domínios organizacionais.

Resposta 3.2:

A integração de serviços heterogéneos em ambientes distribuídos é um desafio fundamental nos sistemas modernos. Neste contexto, o *middleware* baseado em *Web Services* desempenha um papel central, permitindo que componentes desenvolvidos em diferentes plataformas, linguagens ou localizações possam comunicar de forma padronizada. Os dois principais paradigmas utilizados são SOAP (Simple Object Access Protocol) e REST (Representational State Transfer), ambos com abordagens distintas à interoperabilidade e à segurança (Coulouris et al., 2011).

O padrão SOAP é uma especificação formal baseada em XML, com suporte a contratos de interface através de WSDL (Web Services Description Language). Esta abordagem é particularmente robusta em ambientes empresariais complexos, onde são necessários elevados níveis de interoperabilidade, definição rigorosa de serviços e suporte a transações distribuídas. A sua formalização facilita a comunicação entre organizações que utilizam tecnologias muito distintas, como Java EE e .NET, assegurando uma descrição exata das operações e dos tipos de dados. SOAP é frequentemente usado em contextos financeiros, serviços governamentais e sistemas críticos, onde a padronização e o controlo rigoroso das mensagens são prioritários (Alonso et al., 2004).

A nível de segurança, SOAP é compatível com WS-Security, um conjunto de normas que suportam autenticação, assinatura digital, criptografia de mensagens e controle de políticas de acesso. Estas capacidades tornam SOAP adequado para cenários que exigem confidencialidade, integridade e não repúdio. Um exemplo comum é a integração de serviços

bancários ou hospitalares que necessitam de *logs* auditáveis e proteção em profundidade. No entanto, esta segurança tem um custo, uma vez que o processamento XML e a sobrecarga das mensagens SOAP podem impactar negativamente a performance (Coulouris et al., 2011).

Em contraste, o modelo REST não se baseia em XML nem exige contratos formais. É um estilo arquitetural que utiliza os métodos padrão do protocolo HTTP (GET, POST, PUT, DELETE) para operar sobre recursos identificados por URIs. A simplicidade do REST favorece a escalabilidade e o desempenho, sendo amplamente utilizado em APIs públicas e aplicações móveis. A sua interoperabilidade baseia-se em padrões amplamente aceites como JSON e HTTP, o que facilita a adoção rápida em contextos com menor necessidade de formalização. Por exemplo, plataformas como Twitter, GitHub e Google Maps oferecem APIs RESTful para acesso a funcionalidades externas (Newman, 2015).

Em termos de segurança, REST recorre maioritariamente a mecanismos do próprio HTTP, como TLS/SSL, autenticação básica ou *tokens* OAuth. Embora estas soluções sejam adequadas para muitas aplicações, oferecem menor granularidade de controlo em comparação com WS-Security. Assim, REST é mais adequado a cenários com requisitos moderados de segurança e elevado foco na escalabilidade e simplicidade, como integração entre aplicações web ou comunicação entre microserviços (Dragoni et al., 2017).

Em síntese, SOAP é mais indicado para contextos empresariais complexos, onde a interoperabilidade, transações e segurança formal são cruciais. Já REST adapta-se melhor a ambientes dinâmicos e leves, com ênfase na agilidade e compatibilidade generalizada. A escolha entre ambos deve considerar as exigências de interoperabilidade entre sistemas heterogéneos, os requisitos de segurança e o perfil de desempenho esperado.

Referências Bibliográficas 3.2:

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web Services: Concepts, Architectures and Applications. Springer.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara & B. Meyer (Eds.), Present and Ulterior Software Engineering (pp. 195–216). Springer.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

Questão 4.1:

Em sistemas com caches distribuídas (como Redis ou Memcached), quais as estratégias mais eficazes para manter a coerência dos dados entre nós? Compare técnicas baseadas em invalidação, atualização e timeout.

Resposta 4.1:

Em sistemas distribuídos de larga escala, a utilização de caches distribuídas, como Redis ou Memcached, é essencial para reduzir a latência de acesso a dados e diminuir a carga sobre bases de dados centrais. No entanto, essa otimização levanta um desafio crítico, nomeadamente como garantir a coerência dos dados armazenados em múltiplos nós de cache, muitas vezes geograficamente distribuídos, sem comprometer a escalabilidade ou a disponibilidade do sistema (Coulouris et al., 2011).

As três estratégias principais mais adotadas para lidar com esse problema são invalidação, atualização (*write-through* ou *write-behind*) e *timeout* (expiração).

A invalidação consiste em remover ou marcar como desatualizada uma entrada da cache quando ocorre uma alteração na origem dos dados. Esta técnica é eficaz quando se pretende evitar leituras inconsistentes, sobretudo em sistemas que exigem consistência forte, como plataformas de pagamentos ou reservas. Um exemplo prático é o sistema de reservas da CP Comboios de Portugal, onde alterações na disponibilidade de lugares têm de ser refletidas de imediato. A utilização de invalidação ativa garante que os utilizadores não acedem a dados obsoletos, prevenindo reservas duplicadas ou rejeitadas. Contudo, garantir invalidação em tempo real exige mecanismos eficientes de notificação e coordenação entre nós, como *message buses* ou *pub/sub* distribuído, o que pode ser complexo e custoso (Tanenbaum & van Steen, 2007).

A estratégia de atualização, por sua vez, visa propagar a nova informação diretamente para a cache assim que ocorre a alteração na origem. Existem duas variantes, no modelo *write-through*, a escrita ocorre simultaneamente na base de dados e na cache, garantindo sincronismo imediato. No modelo *write-behind*, a cache regista a alteração e adia a escrita na base de dados, o que melhora a performance mas pode introduzir atrasos na persistência. Esta abordagem é comum em sistemas de gestão de sessões como o do GitHub, onde manter os dados atualizados melhora a experiência do utilizador sem comprometer demasiado o desempenho. A desvantagem está na complexidade adicional para garantir integridade entre fontes, especialmente em situações de falha parcial.

A estratégia de *timeout* atribui um tempo de vida (*TTL – time to live*) a cada entrada na cache. Findo esse tempo, o valor é automaticamente removido ou considerado inválido, obrigando a consulta à fonte original. Esta técnica reduz significativamente a complexidade de sincronização, sendo amplamente usada em sistemas que toleram consistência eventual, como catálogos de produtos da Amazon, onde um pequeno atraso na atualização de preços não compromete a operação. No entanto, pode conduzir à entrega temporária de dados obsoletos, o que é inaceitável em contextos críticos.

Na prática, muitos sistemas adotam abordagens híbridas, combinando *timeout* com invalidação ativa em eventos críticos. Por exemplo, o Facebook utiliza caches com TTL para dados não sensíveis, mas aplica invalidação direta via eventos internos para mudanças em perfis e permissões. O Redis, por sua vez, permite configurar diferentes políticas de expiração e integra-se com mecanismos de *keyspace notifications* que permitem avisar aplicações externas quando chaves são modificadas, facilitando a coerência ativa.

A escolha da estratégia ideal depende de múltiplos fatores, incluindo o padrão de leitura/escrita, a criticidade dos dados, o grau de tolerância a incoerências e os requisitos de desempenho. Em sistemas financeiros, onde a integridade é prioritária, invalidação imediata e atualização síncrona são comuns. Já em aplicações de redes sociais ou conteúdos noticiosos, onde o volume de leitura é massivo e a consistência pode ser relaxada, o *timeout* oferece um bom equilíbrio entre simplicidade e eficácia (Alonso et al., 2004).

Em suma, manter a coerência em caches distribuídas é uma tarefa delicada que envolve compromissos entre consistência, desempenho e escalabilidade. As estratégias de invalidação, atualização e *timeout* oferecem diferentes equilíbrios, e a combinação entre elas, de forma contextualizada, permite construir sistemas robustos, eficientes e adequados ao domínio em que operam.

Referências Bibliográficas 4.1:

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). Web Services: Concepts, Architectures and Applications. Springer.

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.

Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

Tanenbaum, A. S., & van Steen, M. (2007). Distributed Systems: Principles and Paradigms (2nd ed.). Pearson Education.

Questão 4.2:

Compare as abordagens baseadas em quorum (como o protocolo de leitura/escrita majoritária) com replicação primária em termos de consistência, disponibilidade e tolerância a falhas em sistemas distribuídos. Em que cenários cada abordagem é mais adequada?

Resposta 4.2:

A replicação de dados é uma técnica fundamental para garantir a continuidade e fiabilidade dos sistemas distribuídos. Ao manter múltiplas cópias da mesma informação em diferentes nós da rede, melhora-se a disponibilidade dos dados e a tolerância a falhas, reduzindo os riscos associados a pontos únicos de falha. As duas estratégias de replicação mais adotadas são a replicação primária e os modelos baseados em quorum. Cada uma apresenta características distintas em termos de consistência, disponibilidade e tolerância a falhas, refletindo os compromissos estabelecidos pelo teorema CAP (Coulouris et al., 2011).

Na replicação primária (também conhecida como mestre-escravo), existe um único nó designado como primário que é responsável por todas as operações de escrita. Os nós secundários (réplicas) podem ser usados para operações de leitura, consoante o grau de consistência requerido. Esta abordagem é particularmente adequada a cenários onde a integridade transacional é crítica e as atualizações devem seguir uma ordem bem definida. Por exemplo, em sistemas de gestão académica, plataformas financeiras ou serviços administrativos, é comum

utilizar MySQL configurado com replicação primária e soluções de alta disponibilidade como MHA (*Master High Availability*) ou *Group Replication*, permitindo que os dados estejam sempre atualizados e ordenados num ponto de controlo central.

Entre as principais vantagens desta abordagem estão a simplicidade na resolução de conflitos e a previsibilidade do comportamento do sistema. Contudo, a existência de um ponto central de escrita introduz riscos significativos de disponibilidade, caso o nó primário falhe. Embora mecanismos de *failover* automático possam atenuar este risco, a comutação entre nós requer coordenação precisa e pode introduzir janelas de indisponibilidade, especialmente em sistemas com elevada taxa de escrita.

Por outro lado, os sistemas baseados em quorum distribuem as responsabilidades de leitura e escrita entre múltiplas réplicas, impondo um número mínimo de confirmações (quorum) para validar cada operação. A condição R + W > N (onde R é o número de réplicas consultadas para leitura, W o número para escrita e N o total de réplicas) assegura que pelo menos uma réplica envolvida numa leitura possui a informação mais atualizada (Tanenbaum & van Steen, 2007). Este modelo melhora substancialmente a disponibilidade, pois permite a continuidade de operação mesmo quando algumas réplicas estão inacessíveis ou com falhas.

Este tipo de abordagem é amplamente adotado em sistemas de armazenamento distribuído, como Ceph ou GlusterFS, muito utilizados em ambientes académicos, plataformas de computação científica e clouds privadas. Nestes contextos, a resiliência a falhas e a escalabilidade horizontal são prioridades, e a consistência pode ser ajustada dinamicamente consoante a criticidade dos dados. A capacidade de operar com partições temporárias da rede e de recuperar automaticamente réplicas desatualizadas faz destes sistemas uma escolha sólida em infraestruturas que exigem elevada robustez.

Apesar das vantagens, os modelos baseados em quorum não são isentos de complexidade. Em redes com elevada latência ou relógios dessincronizados, podem ocorrer leituras inconsistentes ou atrasos na propagação de escritas. Para mitigar estas situações, são frequentemente aplicadas estratégias de resolução de conflitos, como vetores de versão, algoritmos de reconciliação ou políticas de “última escrita válida”. Estas técnicas, embora eficazes, introduzem sobrecarga computacional e requerem lógica adicional na aplicação.

A escolha entre replicação primária e quorum deve, portanto, ser cuidadosamente ponderada em função das necessidades do sistema. A replicação primária é mais indicada em contextos que exigem consistência forte e auditoria rigorosa, como aplicações financeiras ou gestão hospitalar. Já os modelos baseados em quorum são preferíveis quando se pretende maximizar a disponibilidade e a escalabilidade, como em sistemas de armazenamento partilhado, plataformas de serviços web ou ambientes de microserviços.

Em suma, ambas as estratégias oferecem vantagens relevantes no desenho de sistemas distribuídos. Compreender os compromissos inerentes a cada abordagem permite aos arquitetos de sistemas escolher a solução mais adequada aos objetivos operacionais, minimizando riscos e otimizando a performance global do sistema.

Referências Bibliográficas 4.2:

Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). Distributed Systems: Concepts and Design (5th ed.). Addison-Wesley.

Tanenbaum, A. S., & van Steen, M. (2007). Distributed Systems: Principles and Paradigms (2nd ed.). Pearson Education.