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?

A utilização de middlewares orientados a mensagens, como **Apache Kafka** e **RabbitMQ**, desempenham um papel muito importante na construção de **sistemas distribuídos escaláveis e resilientes.**
Eles, os middlewares, ajudam na comunicação assíncrona entre os componentes de um sistema, permitindo que troquem mensagens de forma robusta. Mas apesar disto tudo, a sua utilização envolve desafios relacionados com a **ordem e entrega das mensagens**, requerendo desta forma uma conceção cuidadosa na implementação de projetos desta magnitude.

O **Capítulo 9** sobre **Serviços Web** menciona como os serviços Web permitem interações desacopladas entre clientes e servidores, o que está alinhado com o conceito de **middleware orientado a mensagens**. Em particular, podemos ver que o autor aborda o uso de **mensagens XML** e protocolos como **SOAP**, que possibilitam a troca de informações sem exigir que os participantes estejam ativos simultaneamente. Por exemplo, numa arquitetura de micro-serviços, podemos ter a comunicação feita por meio de filas de mensagens (RabbitMQ) ou tópicos (Kafka), desacoplando a produção e o consumo de dados de forma que cada serviço possa escalar em função das necessidades sem afetar os restantes.

Desafios na Garantia da Ordem e Entrega das Mensagens
Anteriormente referia os benefícios da utilização de middlewares orientados a mensagens, mas garantir **a ordem(i)** e a **entrega(ii)** das mensagens em sistemas distribuídos pode ser desafiador. No Capítulo 10 é destacada a necessidade de sistemas que possam compartilhar dados e recursos em grande escala. Os sistemas peer-to-peer visam lidar com a volatilidade e falhas dos nós participantes.
(i)Ordem das Mensagens
Em sistemas de **alta concorrência**, múltiplos consumidores podem processar mensagens de maneira paralela, o que pode gerar desordem. Para mitigar isso, **Apache Kafka** utiliza:

- **Partições ordenadas**: cada tópico pode ter múltiplas partições, garantindo que mensagens dentro da mesma partição sejam lidas na ordem correta.
- **Chaves de Partição**: permitem agrupar mensagens relacionadas, garantindo que sejam consumidas sequencialmente.

Por outro lado, RabbitMQ utiliza **priorização e roteamento de mensagens**, mas pode sofrer reordenação se múltiplas filas forem processadas simultaneamente.
(ii)Entrega das Mensagens
Conforme discutido no Capítulo 9 sobre **Serviços Web**, a **transmissão confiável de mensagens** é essencial para evitar perdas. As falhas do sistema podem levar à entrega de mensagens duplicadas ou à perda de mensagens. Vários aspetos como:

- **Confirmação de entrega**: em sistemas distribuídos, a entrega de mensagens pode ser comprometida por falhas de rede ou processos interrompidos.
- **Reduplicação de mensagens**: sistemas devem evitar que mensagens sejam entregues mais de uma vez.
- **Persistência das mensagens**: Kafka resolve esse problema armazenando mensagens no disco e replicando-as entre nós.

Imaginemos agora um exemplo do que tentei até agora transmitir - um sistema de processamento de pagamentos que recebe milhares de transações por segundo. Ele precisa garantir:

- Que os pagamentos sejam **processados na ordem correta** (exemplo: Kafka particionando transações por cliente).

- Que nenhuma transação seja **perdida ou duplicada** em caso de falha (exemplo: RabbitMQ exigindo confirmações antes de remover mensagens da fila).

Esse cenário demonstra como os desafios abordados nos **Capítulos 9 e 10** se aplicam na prática, reforçando a importância de middleware orientado a mensagens para **escalabilidade e resiliência**.

Como podemos ver nos capítulos sugeridos pelo professor, nenhuma solução única resolve todos os desafios: a escolha entre Kafka, RabbitMQ ou outras ferramentas depende do perfil de carga, requisitos de ordem, tolerância a falhas e latência aceitável. Testes de stress, monitoramento contínuo e flexibilidade arquitetural são fundamentais para garantir que o sistema responda aos requisitos de negócio e mantenha a integridade dos dados mesmo em cenários de falha.

Em suma, garantir a ordem e a entrega das mensagens permanece o desafio maior da utilização de *middleware* orientado a mensagens (como o Apache Kafka ou RabbitMQ) na construção de Sistemas Distribuídos escaláveis e resilientes, requerendo desta forma atenções redobradas na implementação de projetos desta dimensão.

Bibliografia :

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. *Sistemas distribuídos: conceitos e projeto*. Tradução: João Eduardo Nóbrega Tortello. Revisão técnica: Alexandre Carissimi. 5. ed. Porto Alegre: Bookman, 2013.

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.

A integração de serviços heterogêneos por meio de middleware baseado em Web Services, permite que aplicações de diferentes domínios organizacionais se comuniquem.
SOAP e REST são dois padrões principais que possibilitam essa interoperabilidade, embora com abordagens distintas.
Pelo que li no Capítulo 9, o protocolo SOAP (Simple Object Access Protocol) oferece uma interoperabilidade mais forte e recursos de segurança robustos, mas pode ser mais complexo, ao passo que o protocolo REST (Representational State Transfer) é mais simples, leve e alinhado com a arquitetura da Web, sendo adequado para cenários que exigem escalabilidade e facilidade de uso.
Na página 420 do livro adotado nesta disciplina podemos ler que o SOAP é um protocolo robusto e amplamente utilizado para comunicação entre aplicações distribuídas. Ele foi projetado para permitir que serviços Web troquem informações de maneira segura e padronizada, garantindo interoperabilidade entre diferentes plataformas e linguagens de programação.
Ele estrutura as mensagens em um envelope XML e geralmente utiliza HTTP como transporte, embora outros meios também possam ser empregados. SOAP pode operar de forma síncrona ou assíncrona e permite a inclusão de cabeçalhos opcionais para segurança e extensibilidade.
Embora SOAP seja poderoso, alternativas como **REST** são mais leves e frequentemente preferidas em aplicações modernas, especialmente para serviços que precisam de alta escalabilidade e simplicidade.
REST promove a interoperabilidade ao aderir aos princípios da Web, como URIs para identificar recursos e métodos HTTP padrão (GET, POST, PUT, DELETE) para manipulá-los. A nível de Segurança depende principalmente de protocolos de transporte seguros como **TLS/SSL** e autenticação baseada em **OAuth** e **JWT**, garantindo proteção em arquiteturas abertas.
Na página 384 do livro, temos um exemplo de como os serviços Web da Amazon podem ser consultados tanto por SOAP ou REST, permitindo que desenvolvedores criem aplicações que agregam valor aos serviços da empresa. Isso possibilita, por exemplo, a automação do controle de inventário e pedidos, integrando-se diretamente com a Amazon.com.
Um outro exemplo pratico que consegui descortinar é o caso dos serviços Web do **Google Maps**. A API do Google Maps pode ser acessada tanto por **SOAP** quanto por **REST**, permitindo que desenvolvedores integrem mapas, geolocalização e informações de trânsito nas suas próprias aplicações. Isso possibilita, por exemplo, que um sistema de entrega automatizado calcule rotas eficientes, estime tempos de chegada e otimize a logística com base no trânsito em tempo real.
Esses exemplos ilustram como a adoção de Web Services, por meio de padrões como SOAP e REST oferecem abordagens complementares para garantir interoperabilidade e segurança na integração de serviços heterogêneos. SOAP, com seu protocolo estruturado e extensões de segurança robustas, é ideal para ambientes corporativos que exigem confiabilidade e transações complexas. Já o protocolo REST, mais leve e flexível, prioriza escalabilidade e simplicidade em aplicações Web modernas, utilizando padrões e protocolos seguros da Web, como HTTPS. A escolha entre eles depende das necessidades de cada integração e do contexto organizacional.

Bibliografia:

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. *Sistemas distribuídos: conceitos e projeto*. Tradução: João Eduardo Nóbrega Tortello. Revisão técnica: Alexandre Carissimi. 5. ed. Porto Alegre: Bookman, 2013.

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*.

Confesso que até começar a ler sobre a temática 4, nunca tinha tido contacto com o tema que aborda as estratégias mais eficazes para manter a coerência dos dados entre nós em sistemas com *caches* distribuídas (como Redis ou Memcached).
Pelo que me apercebo, não existem soluções únicas para manter a coerência em caches distribuídas, estando a escolha da estratégia ou combinação de estratégias depende dos requisitos específicos da aplicação em termos de consistência, desempenho e tolerância a falhas.
Penso que a:
(i) Invalidação funciona quando um dado é modificado no sistema principal, a entrada correspondente na cache é marcada como inválida ou removida. Na próxima vez que esse dado for solicitado, ocorrerá um cache miss, e o dado atualizado será buscado no sistema principal e recarregado na cache (geralmente usando a estratégia Read-Through).
(ii) Atualização (Cache Update ou Write-Through/Write-Behind) funciona quando os dados são atualizados na origem, o cache é atualizado imediatamente com os novos valores; entre as suas vantagens podemos ver:
• Mantém o cache sempre atualizado.
• Reduz o número de cache misses.
Um exemplo desta situação são os Sistemas com write-through caching, onde a escrita no banco de dados e no cache ocorre na mesma operação.
(iii)Timeout funciona, como o nome indica, cada entrada na cache tem um tempo de vida (TTL). Após esse período, a entrada é considerada expirada e é removida ou marcada para recarregamento na próxima vez que for acessada (similar ao Read-Through após a expiração), evitando assim, por exemplo, que os dados obsoletos fiquem na cache indefinidamente. Uma desvantagem que pude ver aqui é que definir um TTL muito curto pode levar a muitos cache misses e sobrecarga no sistema principal. Definir um TTL muito longo pode aumentar o período de inconsistência.

Em suma posso fazer um breve resumo entre as diferentes características de cada estratégia:

Estratégia	Consistência	Complexidade	Latência	Ideal para...
Invalidação	Alta	Média	Média	Dados críticos e mutáveis
Atualização	Alta	Alta	Baixa	Sistemas com alta frequência de leitura e escrita
Timeout (TTL)	Baixa/Média	Baixa	Alta	Dados semi-estáticos ou tolerantes a desatualização

Gostaria de partilhar aqui um exemplo que veio à ideia: qual seria a melhor estratégia para desenvolver um contador de visualizações duma página web que é atualizado a cada nova visita? Penso que a melhor estratégia a aplicar seria a **Atualização** porque é necessário refletir cada nova visualização imediatamente para o usuário. Se por outro lado tivesse escolhido a:

- **Invalidação** causaria uma leitura desatualizada até a próxima visita;
- **Timeout** não garantiria a atualização em tempo real. Atualizar a cache a cada visita mantém o contador preciso.

Em suma, a escolha da estratégia de coerência de cache (invalidação, atualização ou timeout) depende criticamente dos requisitos da aplicação quanto à consistência, desempenho e tolerância à desatualização. A **invalidação** prioriza a precisão eventual para dados mutáveis; a **atualização** mantém o cache atualizado em sistemas com alta frequência de leitura/escrita,

mas com maior complexidade; e o **timeout (TTL)** oferece simplicidade para dados semi-estáticos, exigindo cuidado na definição do tempo de vida. A combinação estratégica dessas abordagens pode ser a solução mais eficaz para otimizar o sistema.

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?

Sem dúvida a pergunta mais difícil de responder e para começar a responder, procurei compreender melhor o que significa a abordagem baseada em quorum (como o protocolo de leitura/escrita majoritária) e como ele se aplica a sistemas distribuídos.
Neste site web(1), quorum é descrito como *"o número mínimo de participantes (nós) em um sistema distribuído que deve concordar para validar uma operação ou decisão. Ele é fundamental para garantir a consistência e a confiabilidade das operações em ambientes com múltiplos servidores, especialmente em sistemas com replicação de dados.*" e a sua importância reside "*na sua capacidade de prevenir as ocorrências de falhas de consistência (…) atua como um mecanismo de controle que assegura que apenas as operações que têm o apoio de uma maioria sejam executadas.*". Em suma, quorum é número mínimo de nós que precisam concordar para validar uma operação em sistemas distribuídos e a sua importância destaca-se:

- Prevenção falhas de consistência.
- Garante que apenas operações com apoio da maioria sejam executadas.
- Mecanismo de controle para confiabilidade em ambientes com múltiplos servidores e replicação de dados.

1. A abordagem baseada em **quorum para leitura/escrita** majoritária em sistemas distribuídos consiste em replicar dados em vários nós e exigir que a maioria (quorum) confirme operações de leitura ou escrita. Essa interseção entre os quoruns garante consistência dos dados, mesmo diante de falhas parciais.
As principais vantagens são alta tolerância a falhas e forte consistência, com flexibilidade para ajustar o quorum conforme a prioridade de leitura ou escrita. Entre as desvantagens, destacam-se a possível alta latência das operações e a redução da disponibilidade caso não seja possível atingir o quorum necessário.
2. Ao passo que a **Replicação Primária** funciona como um nó primário que recebe todas as escritas e as replica para nós secundários. (Exemplos: MongoDB, Redis Sentinel…). Dentro das suas vantagens poderemos salientar a baixa latência de escrita e leituras escaláveis (nos secundários); no que concerne as desvantagens podemos ver uma menor tolerância a falhas (comparado ao quorum) e a indisponibilidade durante a eleição de um novo primário em caso de falha.

No que diz respeito aos cenários que cada abordagem é mais adequada, saliento:
Quorum - Ideal para sistemas que exigem forte consistência e alta tolerância a falhas, como bancos de dados distribuídos, sistemas financeiros…
Replicação Primária - Adequada para cenários com alta taxa de leitura e menor taxa de escrita, onde a simplicidade da arquitetura é desejada sendo muito utilizada em bancos de dados relacionais tradicionais.

Em suma, a decisão entre quorum e replicação primária reside no equilíbrio estratégico que se busca alcançar entre consistência, disponibilidade e tolerância a falhas, sempre considerando as necessidades específicas de desempenho e a complexidade de implementação inerentes ao sistema distribuído em questão.

Bibliografia:
(1) https://programae.org.br/software/glossario/o-que-e-quorum-e-como-ele-se-aplica-a-sistemas-distribuidos/