Exame 1

Questão 1 - Concorrência em Sistemas Distribuídos

a) Diferencie programa, processo e thread, descrevendo os principais estados de vida de um processo e de uma thread.

b) Explique em que consiste exclusão mútua e compare o uso de semáforos e de monitores para proteger regiões críticas.

c) Discuta como, em Java, se podem prevenir deadlocks e quais as ferramentas de concorrência de nível elevado disponíveis na plataforma.

a) Ao falarmos de programa, referimo-nos ao ficheiro estático de instruções armazenado em disco. Só quando o sistema operativo carrega esse código na memória, reserva espaços para dados e executa instruções, é que surge o **processo**, estrutura que agrega o espaço de endereçamento, o estado da CPU (registos e contador de instrução) e canais de I/O. Uma **thread** é uma linha de execução dentro desse processo: várias threads num mesmo processo partilham o espaço de memória, mas cada uma mantém a sua stack e registos independentes.

Classicamente, um processo percorre os estados "novo" (após criação), "pronto" (aguarda CPU), "em execução" (a CPU está a executar as suas instruções), "bloqueado" (aguarda evento de I/O ou temporizador) e "terminado". As threads seguem diagrama semelhante, mas quando uma thread termina, o processo pode permanecer vivo se existirem outras threads em execução; e pode existir "timed waiting" quando aguarda um tempo sem estar bloqueada indefinidamente.

b) A **exclusão mútua** garante que apenas uma thread ou processo, entre, de cada vez, numa região crítica, evitando condições de corrida. Os **semáforos** implementam-se como um contador atómico associado a uma fila de espera: a operação P (down) decrementa o contador ou bloqueia se estiver a zero; a operação V (up) incrementa e desperta uma thread. Os semáforos permitem controlar o acesso a múltiplas instâncias de um recurso, mas exigem cuidado para evitar inversões de prioridade ou deadlocks. Já o **monitor** combina um lock interno com variáveis de condição: ao chamar wait() a thread liberta o lock e bloqueia, ficando na fila da variável de condição; ao chamar signal() desperta uma thread dessa fila. O encapsulamento do lock dentro do próprio objeto aumenta a segurança e legibilidade do programa, prevenindo usos incorretos de "lock/unlock".

c) Em **Java**, para prevenir **deadlocks**, pode impor-se uma ordenação total na aquisição de locks, garantindo que todas as threads solicitam recursos na mesma sequência. As bibliotecas de concorrência de nível elevado incluem **ReentrantLock** com múltiplas condições (newCondition()), **CyclicBarrier** (barreiras de sincronização), classes atómicas (por exemplo, AtomicInteger), e

coleções concorrentes (por exemplo, ConcurrentHashMap), que abstraem os detalhes de locks explícitos e reduzem a probabilidade de bloqueios mútuos, aumentando a robustez e a escalabilidade das aplicações.

Questão 2 - Comunicação e Passagem de Mensagens

a) Defina latência, largura de banda e jitter, explicando o seu impacto na comunicação entre nós distribuídos.

b) Diferencie primitivas síncronas versus assíncronas e operações bloqueantes versus não-bloqueantes num modelo de passagem de mensagens.

c) Compare os protocolos TCP e UDP no contexto de sockets, apontando vantagens, desvantagens e cenários de utilização adequados.

a) A **latência** mede o atraso entre o instante em que o emissor inicia o envio de dados e o momento em que o recetor recebe o primeiro bit. A **largura de banda** (bandwidth) quantifica o volume total de dados que pode ser transmitido por unidade de tempo. Já o **jitter** traduz a variabilidade no atraso de entrega das mensagens, sendo crítico em aplicações em tempo real como voz sobre IP ou vídeo, pois grandes variações provocam cortes e degradações percetíveis. Em sistemas distribuídos, latências elevadas tornam mais cara a sincronização global, enquanto largura de banda limitada reduz o débito de dados e jitter excessivo obriga a buffers maiores, agravando a latência aparente.

b) Nas primitivas de comunicação, uma operação **síncrona** só retorna ao chamador quando existe garantia de que o recetor já realizou a operação complementar (por exemplo, que já fez o receive correspondente); numa interface **assíncrona**, o send devolve logo que copia os dados para um buffer local, deixando a conclusão da transmissão pendente. Quanto a bloqueios, numa chamada **bloqueante** o processo fica suspenso até a operação completar, enquanto numa **não-bloqueante** a chamada devolve de imediato, disponibilizando um handle ou estado que o programa consulta mais tarde para verificar o progresso.

c) O **TCP** (Transmission Control Protocol) é orientado a conexão, fiável e bidirecional: estabelece um canal virtual com handshake de três vias, garante entrega ordenada e controlo de fluxo/erros, sendo ideal para HTTP, SSH ou transferência de ficheiros, onde a perda de pacotes ou a reordenação são inaceitáveis. Contudo, o overhead do estabelecimento de conexão e do controlo de congestionamento introduz latência extra. O **UDP** (User Datagram Protocol) é **connectionless**, sem garantias de entrega nem ordenação, oferecendo maior velocidade e menos overhead, adequado a streaming multimédia, DNS ou jogos em tempo real, onde perdas pontuais podem ser toleradas, mas a latência reduzida é

essencial. Num socket TCP, o servidor cria um listening socket e cada conexão gera um novo socket de comunicação; em UDP, apenas existe um socket de envio e receção, e cabe à aplicação implementar eventuais mecanismos de fiabilidade se necessário.

Questão 3 - Sincronização e Relógios

a) Descreva os métodos de Cristian e de Berkeley para sincronizar relógios físicos, mencionando vantagens e limitações de cada um.

b) Explique como funciona o Network Time Protocol (NTP) e por que é mais resiliente em ambientes heterogéneos.

c) Compare os relógios lógicos de Lamport com os relógios vetoriais, indicando como cada um captura noção de causalidade.

a) O **método de Cristian** baseia-se num modelo cliente-servidor de tempo: o cliente envia um pedido ao servidor considerado referência UTC e, ao receber a resposta, calcula o desvio entre o seu relógio e o horário indicado, subtraindo metade do tempo de ida e volta (RTT). Se a rede for estável e simétrica, o ajuste aproxima eficazmente o relógio local do UTC, mantendo monotonicidade. Porém, em redes instáveis, as variações de RTT introduzem erros de estimação que podem desestabilizar o sincronismo, obrigando o cliente a ignorar as rondas com atrasos anómalos.

No **algoritmo de Berkeley**, não existe servidor UTC; um dos nós assume o papel de mestre e periodicamente interroga todos os outros, incluindo a si próprio, para recolher leituras dos relógios e estimar atrasos. Calcula então a média dos desvios e envia a cada nó a correção diferencial a aplicar. Como as correções são relativas e centradas na mediana do conjunto, consegue-se sincronismo interno consistente, mas o tempo resultante pode divergir do UTC real. É útil em sistemas fechados onde não há acesso a servidores externos de maior precisão.

b) O **Network Time Protocol (NTP)** organiza servidores em estratos hierárquicos: no estrato 0 os relógios atómicos ou GPS; no estrato 1, servidores que se ligam diretamente aos estratos superiores; e assim por diante. Em cada troca, dois nós A e B trocam quatro carimbos de tempo (timestamps) - instantes de envio e receção de cada mensagem - e calculam tanto o offset do relógio como a assimetria do caminho. Repetindo o procedimento e filtrando estatisticamente as amostras, um cliente consegue manter sincronismo dentro de dezenas de milissegundos, mesmo atravessando múltiplas redes e estratos, o que torna o NTP extremamente resiliente a falhas pontuais e heterogeneidade de infraestruturas.

c) Os **relógios lógicos de Lamport** mantêm um contador escalar que incrementa a cada evento local e ao enviar mensagens anexa o seu valor. Ao receber, o processo atualiza o seu contador para o máximo entre o próprio e o timestamp recebido, e só depois incrementa. Isto garante que, se um evento *e* preceder causalmente *e′*, então timestamp(*e*) < timestamp(*e′*), capturando a noção de "happened-before", mas sem distinguir concorrência de causalidade direta. Já os **relógios vetoriais** fazem cada processo manter um vetor com N entradas (uma por processo). Em cada evento, incrementa apenas a sua componente; ao receber, funde componente a componente pelo máximo e depois incrementa a sua entrada. Assim, é possível determinar se dois eventos são comparáveis (e, portanto, há causalidade) ou concurrentes (vetores incomparáveis), permitindo rastrear precisamente dependências de causalidade no sistema.

Questão 4 - Transações Distribuídas

a) Explique as propriedades ACID e justifique a necessidade de coordenar transações em vários nós.

b) Descreva o protocolo de Duas Fases (2P**C)** e identifique o problema de bloqueio que pode surgir.

c) Apresente o protocolo de Três Fases (3P**C)** e as transações aninhadas como soluções para mitigar bloqueios e aumentar modularidade.

a) As transações obrigam a que um conjunto de operações de leitura e escrita seja tratado como atómico: ou todas ocorrem com sucesso, ou nenhuma se concretiza. O acrónimo **ACID** sintetiza estas garantias: **Atomicidade** (tudo ou nada), **Consistência** (passar de um estado válido a outro sem violar invariantes), **Isolação** (execução concorrente sem ver estados intermédios de outras transações) e **Durabilidade** (efeitos persistem mesmo perante falha). Em sistemas distribuídos, dados de uma mesma transação podem residir em máquinas distintas, exigindo um coordenador global para assegurar que cada fragmento confirma o commit ou aborta em conjunto, de modo a manter a coerência de ponta a ponta.

b) O **protocolo de Duas Fases (2PC)** decorre em duas fases. Na fase de votação, o coordenador envia "prepare" a cada participante, que responde "voteCommit" (se puder confirmar) ou "voteAbort". Na fase de decisão, se todos votarem a favor, o coordenador envia "commit" a todos; caso contrário envia "abort". O problema central do 2PC é o **bloqueio**: se o coordenador falhar após recolher votos favoráveis, mas antes de enviar a decisão, os participantes ficam indefinidamente à espera, mantendo locks sobre recursos e impedindo progresso.

c) O **protocolo de Três Fases (3PC)** introduz uma fase intermédia de "pre-commit": depois de receber votos favoráveis, o coordenador envia um aviso de pré-compromisso para que os participantes entrem num estado preparado, mas não confirmem ainda. Só após obter confirmações desse estado é que envia o "commit" final. Este passo extra permite, em caso de falha do coordenador, que os participantes consultem uns aos outros e decidam unicamente, evitando bloqueios duradouros. Já as **transações aninhadas** organizam-se hierarquicamente: uma transação principal dispara subtransações, cada uma com commit local tentativo. Se a transação de nível superior abortar, todas as subtransações em cascata também abortam; se for bem-sucedida, então todos os commits locais tornam-se definitivos. Este modelo permite paralelizar e modularizar fluxos de trabalho, reduzindo a superfície de bloqueios e focalizando a recuperação de falhas em subcomponentes sem afetar todo o sistema.

Exame 2

Questão 1 - Modelos de Sistemas Distribuídos

a) Compare os modelos arquiteturais de sistemas distribuídos: cliente-servidor, peer-to-peer e publisher-subscriber.

b) Explique o papel do broker no modelo publisher-subscriber e como ele promove o desacoplamento entre produtores e consumidores.

c) Dê exemplos concretos de aplicações que se encaixem em cada um destes modelos, justificando a escolha.

a) O **modelo cliente-servidor** é caracterizado por uma separação clara de papéis: o cliente requisita serviços e o servidor centraliza a lógica e o controlo dos recursos. Este modelo favorece o controlo, segurança e gestão centralizada, mas sofre de pontos únicos de falha e limitações de escalabilidade. Já o **modelo peer-to-peer (P2P)** distribui a responsabilidade por todos os nós, que podem atuar tanto como clientes como como servidores. Há partilha de recursos e colaboração sem hierarquia, promovendo tolerância a falhas e escalabilidade, mas dificultando a gestão de consistência e segurança.

a) e **b)** O **modelo publisher-subscriber** introduz um intermediário - o **broker** - que separa os produtores (publishers) dos consumidores (subscribers). Ao invés de enviar mensagens diretamente, os produtores publicam dados num canal ou tópico, e o broker encaminha apenas para os subscritores interessados. Este modelo promove o **desacoplamento espacial, temporal e de sincronização**: os

publishers não precisam conhecer os consumidores, estes podem ligar-se mais tarde e o broker gere a entrega, podendo persistir, filtrar ou transformar mensagens. Tal arquitetura é ideal para sistemas assíncronos e escaláveis, como em aplicações IoT ou de eventos complexos.

c) Exemplos concretos incluem sistemas de **email ou web servers** no modelo cliente-servidor, onde há pedido-resposta bem definido; redes de **partilha de ficheiros como BitTorrent** ou sistemas blockchain que operam em P2P para resistência a falhas e descentralização; e plataformas de **notificações push, MQTT ou Apache Kafka** como casos de uso do modelo publisher-subscriber, em que sensores publicam leituras e consumidores subscrevem apenas os dados relevantes.

Questão 2 - Modelos Fundamentais de Sistemas Distribuídos

a) Descreva os modelos fundamentais de interação, falhas e segurança em sistemas distribuídos.

b) Diferencie falhas por omissão, temporização e bizantinas, explicando os desafios na deteção e recuperação.

c) Quais são as ameaças típicas à segurança em processos e canais? Como se implementa controlo de acesso?

a) Os **modelos fundamentais** de sistemas distribuídos descrevem propriedades essenciais do ambiente de execução. O **modelo de interação** aborda aspetos como **latência**, **largura de banda** e **jitter**, os quais afetam a comunicação entre processos e impõem limites à sincronização. O **modelo de falhas** classifica os tipos de erro que podem ocorrer em processos ou canais - desde falhas simples até comportamentos arbitrários - e define os pressupostos necessários para construir algoritmos robustos. Já o **modelo de segurança** foca-se em garantir **confidencialidade, integridade e autenticidade** perante ameaças de terceiros ou processos maliciosos.

b) As **falhas por omissão** ocorrem quando uma mensagem ou resposta esperada não chega ao destino; são relativamente fáceis de detetar com timeouts. **Falhas por temporização** ocorrem quando respostas chegam fora da janela temporal aceite, podendo confundir processos síncronos. As **falhas bizantinas** são as mais complexas, pois implicam comportamento arbitrário, incluindo envio de mensagens inconsistentes a diferentes nós. A tolerância a estas falhas exige protocolos como os algoritmos de consenso bizantino, que assumem uma fração limitada de nós maliciosos e requerem comunicações redundantes para garantir consistência.

c) As ameaças à **segurança** incluem a **interceção e modificação de mensagens nos canais** e a **impersonação de processos legítimos**. Para mitigar estes riscos, é necessário autenticar processos, cifrar comunicações com chaves simétricas ou assimétricas e usar canais seguros como TLS. O **controlo de acesso** define políticas sobre quem pode aceder a que recursos, e implementa-se por mecanismos como **listas de controlo de acesso (ACLs)** ou **capabilities**, que atribuem permissões específicas a identificadores seguros. A segurança robusta exige também gestão cuidadosa de chaves e monitorização contínua.

Questão 3 - Execução Remota e Invocação Distribuída

a) Explique os princípios de invocação remota, incluindo a função dos stubs e do processo de marshaling.

b) Como funciona a comunicação entre cliente e servidor numa arquitetura com invocação remota?

c) Qual o papel da serialização Java neste contexto e como simplifica o desenvolvimento distribuído?

a) A **invocação remota** permite que um processo invoque métodos ou funções de objetos localizados noutro processo (normalmente noutro nó), como se fossem locais. Para tal, é necessário criar um **stub** (ou proxy) no lado do cliente, que expõe a interface remota e reencaminha chamadas para o lado do servidor. O stub atua como um emulador do objeto remoto. O processo de **marshaling** consiste na conversão dos parâmetros da chamada (e eventualmente do resultado) num formato transmissível - por exemplo, um fluxo de bytes -, que possa ser enviado pela rede. O **unmarshaling** é o processo inverso, no lado recetor.

b) Numa arquitetura cliente-servidor com invocação remota, o **cliente chama um método no stub**, que serializa os dados, envia uma mensagem pela rede, e aguarda resposta. No lado do servidor, um **proxy-agent** desembrulha os dados, invoca o método real, embala o resultado e devolve ao cliente. Esta comunicação pode ser síncrona (cliente esper**a)** ou assíncrona (cliente continua e recebe callback mais tarde). A responsabilidade de garantir que a chamada aparenta ser local, com transparência de localização e comunicação, recai sobre o middleware.

c) A **serialização em Java** converte objetos em fluxos de bytes automaticamente, desde que implementem a interface Serializable. Isto permite simplificar a lógica de marshaling: objetos complexos podem ser passados por valor sem que o programador tenha de escrever manualmente código para os embalar. Esta funcionalidade é essencial em frameworks como RMI (Remote Method Invocation), onde o stub gera e processa mensagens estruturadas que encapsulam chamadas,

resultados e exceções. A serialização facilita assim o desenvolvimento distribuído modular e transparente.

Questão 4 - Consistência, Replicação e Eleição de Líder

a) Descreva os diferentes modelos de consistência aplicáveis a sistemas distribuídos com dados replicados.

b) Compare os protocolos de exclusão mútua baseados em token, votação e relógios lógicos.

c) Explique como funciona um algoritmo de eleição de líder em anel e quais as garantias que deve oferecer.

a) Num sistema com **replicação de dados**, a consistência define o comportamento esperado das leituras e escritas. A **consistência estrita** é o ideal teórico onde cada leitura devolve o valor mais recente, como se existisse uma memória global instantânea, mas é irrealista em redes distribuídas devido à latência. A **linearizabilidade** relaxa ligeiramente esse modelo, impondo apenas ordem respeitante ao tempo real. A **consistência sequencial** exige que todas as operações sejam vistas por todos na mesma ordem, mas não necessariamente em tempo real. Já a **consistência causal** respeita as relações de causa e efeito entre operações (por exemplo, se B depende de A, A deve ser visível antes de B), mas permite intercalar eventos independentes. Finalmente, a **consistência FIFO** garante apenas que as escritas de um mesmo processo são vistas por todos na ordem emitida, sendo o modelo mais fraco, mas eficiente.

b) Para coordenar acesso mútuo a recursos partilhados, há vários **protocolos de exclusão mútua**. O **modelo com token circulante** usa um anel lógico onde o processo que possui o token tem permissão para entrar na secção crítica. É eficiente (uma mensagem por entrad**a)** mas vulnerável à perda do token. O **protocolo de Ricart & Agrawala** baseia-se em **relógios lógicos de Lamport**: os processos enviam pedidos com timestamp e aguardam confirmações antes de avançar. Garante ordem total, mas envolve N mensagens por entrada. O **algoritmo de Maekawa** divide os processos em subconjuntos que se intersectam: cada processo só precisa de votos da sua maioria, reduzindo mensagens a $\sqrt{n}$, mas aumenta complexidade de falhas e recuperação.

c) Num **algoritmo de eleição de líder em anel**, cada processo conhece apenas o seu vizinho e pode iniciar a eleição ao detetar falha do coordenador. A mensagem de eleição circula com o identificador do processo mais forte (por exemplo, o com maior ID), e após uma volta completa, o novo líder é proclamado. As **garantias essenciais** são: **terminação** (todos conhecem o líder ao fim de tempo finito),

univocidade (um só processo é eleito) e **consenso** (todos concordam sobre quem venceu). O algoritmo deve ainda prever recuperação de falhas, como reeleição após timeout ou retransmissão de mensagens perdidas.

Exame 3

Questão 1 - Tempo e Relógios Lógicos

a) O que significa "tempo lógico" num sistema distribuído? Porque não é viável utilizar tempo físico global?

b) Explique como os relógios de Lamport capturam a relação de precedência entre eventos.

c) Em que situações os relógios vetoriais são preferíveis aos de Lamport? Que informação extra permitem obter?

a) O conceito de **tempo lógico** surge para lidar com a ausência de um relógio global perfeitamente sincronizado em sistemas distribuídos. Em vez de confiar no tempo físico - sujeito a **desvios de relógios**, **latência de rede** e **jitter** -, o tempo lógico visa capturar a **ordem causal** entre eventos, permitindo raciocinar sobre "o que ocorreu antes de quê" sem depender de valores absolutos. Isto é fundamental para garantir consistência em replicações e para resolver conflitos em execuções concorrentes.

b) Os **relógios de Lamport** representam o tempo lógico com um simples contador inteiro. Cada processo incrementa o contador a cada evento local e, quando envia uma mensagem, anexa esse valor. Ao receber uma mensagem, o processo atualiza o seu contador para o máximo entre o seu valor atual e o recebido, e só depois o incrementa. Desta forma, se um evento e causar outro e', então $L(e) < L(e')$. Este mecanismo permite ordenar eventos de forma a respeitar a relação de "happened-before".

c) Contudo, os relógios de Lamport não distinguem entre eventos **concorrentes** (sem relação causal) e eventos **causalmente relacionados**, pois podem atribuir timestamps iguais a eventos diferentes. Já os **relógios vetoriais** permitem essa distinção: cada processo mantém um vetor com um contador por processo. Ao trocar mensagens, fundem-se vetores componente a componente. Assim, se os vetores de dois eventos forem incomparáveis (nenhum é menor ou igual ao outro), os eventos são concorrentes. Esta capacidade é crucial para sistemas onde se precisa de detetar **causalidade explícita**, como na manutenção de consistência causal em bases de dados distribuídas.

Questão 2 - Consistência em Sistemas Replicados

a) Porque é que a consistência estrita é impraticável em sistemas distribuídos?

b) Diferencie linearizabilidade, consistência sequencial e causal. Dê exemplos conceituais de situações onde se aplicam.

c) Como é que o uso de relógios lógicos ajuda a implementar consistência causal?

a) A **consistência estrita** exige que todas as operações de leitura devolvam sempre o valor mais recente escrito, como se existisse uma única cópia global e atómica dos dados. Para cumprir esta exigência, seria necessário sincronizar instantaneamente todos os nós após cada escrita, o que é impossível na prática devido às **latências não determinísticas da rede**, **jitter** e **ausência de relógio global preciso**. Além disso, atrasos e falhas podem impedir que todas as réplicas vejam as atualizações no mesmo instante, tornando a consistência estrita inviável.

b) A **linearizabilidade** é uma forma mais fraca, mas ainda forte de consistência: exige que todas as operações pareçam ocorrer em uma única linha temporal que respeita a ordem real (wall-clock). Se uma operação ocorre antes de outra, o sistema deve refletir isso. A **consistência sequencial**, por outro lado, exige apenas que todos vejam as operações numa mesma ordem total que preserve a ordem dos programas individuais - mas esta ordem não precisa coincidir com o tempo real. Já a **consistência causal** garante apenas que, se uma operação influenciar outra (por exemplo, uma resposta depende de uma pergunta), então a ordem é respeitada; eventos sem ligação causal podem ser vistos em ordens diferentes.

Por exemplo, em **linearizabilidade**, um sistema bancário que recebe depósitos e transferências deve manter a ordem real para evitar saldo incorreto. Na **consistência sequencial**, pode-se tolerar leituras ligeiramente desatualizadas desde que a ordem geral seja coerente. Já na **consistência causal**, uma publicação num fórum deve sempre ser visível antes das respostas, mas outras publicações independentes podem surgir em qualquer ordem.

c) Os **relógios lógicos vetoriais** ajudam a implementar consistência causal porque registam com precisão a ordem causal entre eventos. Ao propagar vetores com atualizações, os nós conseguem saber se já aplicaram todas as operações causais necessárias antes de aplicar uma nova, evitando violar relações de dependência entre eventos.

Questão 3 - Comunicação Assíncrona e Middleware

a) Quais são os principais desafios da comunicação entre processos distribuídos?

b) Compare primitivas síncronas e assíncronas e explique como o middleware atua para esconder complexidade ao programador.

c) Qual a importância do uso de sockets e protocolos como TCP e UDP nesse contexto?

a) A **comunicação em sistemas distribuídos** enfrenta diversos desafios, incluindo **latência imprevisível**, **falhas de rede**, **perda ou duplicação de mensagens**, e **ausência de relógios sincronizados**. Estes fatores dificultam garantir entrega de mensagens, ordem correta e deteção de falhas. Além disso, os sistemas podem ser heterogéneos, com diferentes arquiteturas, sistemas operativos ou linguagens, o que complica a interoperabilidade direta entre processos.

b) As **primitivas de comunicação** podem ser **síncronas**, em que o emissor bloqueia até que o recetor confirme a receção, ou **assíncronas**, onde o envio é imediato e a entrega ocorre mais tarde. Esta distinção afeta o modelo de programação: a sincronização simplifica raciocínio, mas limita concorrência; a assíncrona aumenta desempenho, mas exige callbacks ou buffers para gerir a comunicação pendente.

O **middleware** atua como camada intermédia entre aplicações e rede, abstraindo detalhes como serialização de dados, localização de nós, protocolo de transporte, autenticação e retransmissão. Ferramentas como CORBA, Java RMI, gRPC ou RESTful APIs com JSON/HTTP são exemplos de middleware que fornecem **interfaces estáveis e portáveis**, permitindo ao programador focar-se na lógica da aplicação sem lidar com os detalhes baixos da rede.

c) Os **sockets** representam pontos finais de comunicação, identificados por um endereço IP e um porto. Em aplicações baseadas em **TCP**, o socket estabelece uma conexão fiável, com ordem e correção de erros. Em **UDP**, não há conexão nem garantias de entrega, sendo mais rápido e leve. A escolha do protocolo depende da aplicação: TCP para transferência de dados críticos, UDP para aplicações em tempo real. Ambos os protocolos são base da comunicação em middleware distribuído, permitindo a construção de canais lógicos por cima da infraestrutura física da Internet.

Questão 4 - Exclusão Mútua e Eleição de Líder

a) Por que a exclusão mútua é mais difícil em sistemas distribuídos do que em sistemas locais?

b) Compare os algoritmos de exclusão mútua centralizado, token ring e Ricart & Agrawala.

c) O que garante que um algoritmo de eleição de líder seja correto e completo?

a) A **exclusão mútua** visa garantir que apenas um processo de cada vez acede a uma secção crítica partilhada. Em sistemas locais, isso pode ser feito com locks e semáforos apoiados em memória partilhada. Contudo, em **sistemas distribuídos**, os processos estão em máquinas distintas e comunicam via mensagens, o que introduz **latência**, **incerteza na entrega**, **possibilidade de falhas** e **ausência de memória partilhada**. Isso torna mais complexo garantir acesso exclusivo de forma eficiente e tolerante a falhas.

b) O **algoritmo centralizado** designa um coordenador que gere os pedidos: simples e eficiente, mas com ponto único de falha. O **token ring** organiza os processos em anel lógico, passando um token de permissão. É eficiente e evita mensagens desnecessárias, mas depende da integridade do anel e do token. O **algoritmo de Ricart & Agrawala** é descentralizado: cada processo envia pedidos a todos os outros e espera respostas antes de entrar na secção crítica. Garante ordem com base em relógios lógicos, mas envolve maior número de mensagens e bloqueia se um processo falhar sem responder.

c) Um **algoritmo de eleição de líder** é correto se satisfizer três propriedades: **terminação** (a eleição termina em tempo finito), **univocidade** (só um processo é escolhido) e **acordo** (todos concordam no mesmo vencedor). Para alcançar estas propriedades, o algoritmo deve considerar falhas (por exemplo, detetando inatividade), empates (usando IDs únicos) e garantir que nenhuma eleição ambígua ocorre simultaneamente. A completude do algoritmo depende da comunicação fiável e da recuperação após falhas para que o sistema mantenha sempre um líder ativo e conhecido.

Exame 4

Questão 1 - Organização Multithreading

a) Qual é a principal vantagem de usar multithreading em aplicações distribuídas?

b) Compare modelos de multithreading baseados em processos, kernel threads e user-level threads.

c) Como a concorrência multithreaded se relaciona com a comunicação em cliente-servidor?

a) A utilização de **multithreading** permite que uma aplicação distribua o seu trabalho por múltiplas unidades de execução concorrentes, chamadas threads,

que partilham o mesmo espaço de memória. Isto é especialmente útil em sistemas distribuídos porque melhora a **responsividade** e **paralelismo**, permitindo tratar múltiplos pedidos remotos em simultâneo, reduzir latência de comunicação e utilizar melhor os recursos da máquina.

b) Num modelo baseado em **processos**, cada unidade de execução é isolada, com memória independente. É mais seguro, mas mais pesado em termos de contexto e custo de comunicação entre processos. Já as **kernel threads** são geridas pelo sistema operativo, que faz o escalonamento diretamente. São eficientes e bem integradas com os recursos do sistema, mas têm overhead por cada troca de contexto. As **user-level threads**, por sua vez, são geridas por bibliotecas da linguagem, com mudanças rápidas de contexto, mas não aproveitam CPUs múltiplas sem suporte nativo, e bloqueios de uma thread podem afetar todas.

c) Na arquitetura **cliente-servidor**, o servidor pode usar uma thread por ligação (modelo one-thread-per-request), permitindo que múltiplos clientes sejam atendidos simultaneamente. Isso evita bloqueios enquanto espera dados ou resposta de rede, e aumenta o throughput total da aplicação distribuída. Assim, o uso inteligente de multithreading torna-se uma componente essencial da escalabilidade e eficiência.

Questão 2 - Modelos de Interação e Comunicação

a) O que são latência, largura de banda e jitter? Como influenciam o desempenho de um sistema distribuído?

b) Como a variação de jitter afeta aplicações em tempo real? Que estratégias podem ser usadas para mitigá-lo?

c) De que forma a escolha entre primitivas síncronas e assíncronas impacta o design do sistema?

a) A **latência** é o tempo que decorre entre o envio de uma mensagem e a sua receção; **largura de banda** é a quantidade máxima de dados que pode ser transmitida por unidade de tempo; e o **jitter** é a variação da latência ao longo do tempo. Estes três parâmetros definem a qualidade da comunicação num sistema distribuído. Latência elevada aumenta o tempo de resposta; largura de banda limitada reduz o débito máximo; jitter torna imprevisível o desempenho, o que é especialmente problemático para aplicações sensíveis a tempo.

b) Aplicações em **tempo real**, como chamadas de vídeo ou sensores industriais, exigem entrega regular de pacotes. Se o jitter for elevado, os pacotes podem chegar desordenados ou demasiado tarde, levando a interrupções ou falhas. Para mitigar o jitter, usam-se técnicas como buffers de suavização, controlo de tráfego na rede,

ou protocolos como RTP sobre UDP, que priorizam a regularidade sobre a fiabilidade.

c) As **primitivas síncronas** bloqueiam o emissor até que o recetor processe a mensagem, facilitando a lógica de programação, mas reduzindo o desempenho e paralelismo. Já as **primitivas assíncronas** libertam o emissor após envio, permitindo melhor escalabilidade e reatividade. A escolha entre elas depende do equilíbrio entre **simplicidade, desempenho e tolerância a falhas** que se pretende alcançar.

Questão 3 - Modelos de Falhas e Tolerância

a) Diferencie falhas de omissão, temporização e bizantinas em sistemas distribuídos.

b) Porque é que falhas bizantinas são particularmente desafiadoras?

c) Quais estratégias existem para detetar e mitigar falhas nos canais e processos?

a) As **falhas por omissão** ocorrem quando um componente deixa de responder ou de transmitir mensagens esperadas. São fáceis de detetar com mecanismos como timeouts. **Falhas de temporização** envolvem respostas que chegam fora do tempo esperado, típicas em sistemas com requisitos temporais. Já as **falhas bizantinas** são aquelas em que o componente falha de maneira arbitrária, podendo enviar informações incorretas ou inconsistentes para diferentes destinos - sendo as mais difíceis de lidar.

b) A dificuldade com as **falhas bizantinas** reside na impossibilidade de distinguir comportamento malicioso de falhas inocentes, e na necessidade de consenso entre processos, mesmo quando alguns deles podem mentir ou sabotar deliberadamente o sistema. Isso requer algoritmos como **PBFT (Practical Byzantine Fault Tolerance)**, que usam redundância e quorum para tomar decisões seguras, ao custo de maior complexidade e comunicação.

c) Para **detetar e mitigar falhas**, usam-se **watchdogs**, **heartbeat messages**, **timeout adaptativo**, e **retransmissões** automáticas. Técnicas como **replicação ativa**, **checkpoint/recovery**, e **quorum de decisão** aumentam a resiliência. A tolerância a falhas requer um modelo claro de quais falhas são esperadas e uma arquitetura capaz de continuar operação mesmo em presença parcial de erros.

Questão 4 - Transações Aninhadas

a) O que são transações aninhadas e como se organizam hierarquicamente?

b) Como a atomicidade e isolamento são geridos em transações com múltiplos subníveis?

c) Quais as vantagens das transações aninhadas sobre transações planas em sistemas distribuídos?

a) As **transações aninhadas** são uma extensão do modelo de transação tradicional, permitindo que uma transação principal seja composta por **subtransações**, organizadas em forma de árvore. Cada subtransação executa operações que podem ser **confirmadas localmente** (commit parcial), mas só se tornam definitivas quando a transação principal é confirmada. Se a transação superior abortar, todas as subtransações devem também abortar, assegurando atomicidade.

b) A **atomicidade** numa transação aninhada é garantida de forma hierárquica: um commit local não é visível fora do nível até que o commit global ocorra. O **isolamento** pode ser configurado de forma mais flexível: subtransações podem ser visíveis apenas dentro do contexto do pai, evitando interferência externa. Isso permite paralelismo e modularidade, mas exige controlo rigoroso de dependências e acesso a dados partilhados.

c) As principais **vantagens** das transações aninhadas incluem maior **modularidade**, **reutilização de lógica**, **tolerância a falhas localizada** (falhas numa subtransação podem ser isoladas e recuperadas sem invalidar a transação principal), e melhor **controlo de concorrência**, permitindo que subcomponentes de um sistema operem de forma semi-independente. Isso torna as transações aninhadas ideais para fluxos de trabalho distribuídos complexos.

Exame 5

Questão 1 - Comunicação e Sockets

a) Explique a diferença entre comunicação orientada a conexão e sem conexão, dando exemplos com TCP e UDP.

b) Como os sockets permitem que processos se comuniquem? O que identifica exclusivamente um socket?

c) Porque é que UDP é preferido em certas aplicações, apesar de não garantir fiabilidade?

a) Na **comunicação orientada a conexão**, como no protocolo TCP, os processos estabelecem uma ligação estável, negociada com um handshake inicial, onde cada

mensagem tem garantia de entrega, ordenação e integridade. Já a comunicação **sem conexão**, como no protocolo UDP, não requer esta negociação: as mensagens (datagramas) são enviadas diretamente, podendo ser perdidas, duplicadas ou chegar fora de ordem.

b) Um **socket** é um ponto final de comunicação. No contexto IP, é identificado por um par composto pelo **endereço IP** e o **número de porta**. Um socket TCP também inclui o protocolo e pode ter um estado de conexão associado. Através de chamadas como send() e recv(), os processos trocam dados encapsulados nesses sockets.

c) Apesar de não garantir fiabilidade, o **UDP** é preferido quando a latência é mais crítica que a exatidão - por exemplo, em jogos online, chamadas de voz ou streaming de vídeo - onde pequenas perdas são aceitáveis e a retransmissão causaria atrasos mais prejudiciais que a falha em si.

Questão 2 - NTP, Cristian e Berkeley

a) Explique como o NTP atinge sincronização precisa em redes públicas e distribuídas.

b) Compare os métodos de Cristian e Berkeley. Quando é que cada um é mais adequado?

c) Quais são as limitações inerentes à sincronização em sistemas distribuídos?

a) O **Network Time Protocol (NTP)** usa uma hierarquia de servidores de tempo (estratos), onde os de nível superior são ligados a fontes muito precisas, como GPS ou relógios atómicos. Cada nó na rede consulta múltiplos servidores e calcula o **offset de tempo e o atraso da rede** com base em quatro timestamps: envio, receção, resposta e regresso. Algoritmos de filtragem eliminam amostras irregulares, permitindo que os relógios locais se ajustem suavemente, mantendo erros abaixo de dezenas de milissegundos.

b) O **método de Cristian** depende de um servidor UTC. O cliente envia um pedido e estima o tempo atual do servidor subtraindo metade do tempo de ida e volta. É simples, mas assume que a latência é simétrica. O **algoritmo de Berkeley** não exige fonte externa: um mestre consulta os relógios dos outros, calcula a média e envia a cada um o ajuste a aplicar. É útil em sistemas fechados que não precisam de sincronização com tempo real.

c) A sincronização é sempre limitada por **jitter**, **latência assimétrica**, e **desvios de relógio locais (clock drift)**. Além disso, falhas na rede ou nos próprios relógios

podem provocar discrepâncias que não são completamente evitáveis, apenas minimizadas.

Questão 3 - Sincronização e Barreira

a) O que é uma barreira de sincronização? Em que contexto se utiliza?

b) Como monitores e semáforos são usados para coordenar múltiplos processos?

c) Qual a vantagem de usar ferramentas de alto nível como CyclicBarrier ou CountDownLatch em Java?

a) Uma **barreira de sincronização** é um ponto de execução onde múltiplos processos ou threads devem esperar até que todos tenham chegado, antes de continuar. É utilizada quando tarefas paralelas devem estar sincronizadas num ponto comum, como no final de uma fase de computação paralela.

b) Monitores encapsulam métodos sincronizados com variáveis de condição, permitindo que apenas uma thread execute dentro do monitor, enquanto outras esperam. **Semáforos** usam um contador interno para permitir acesso a recursos limitados ou sincronizar múltiplos processos com P() e V() (wait/signal).

c) Em **Java**, ferramentas como **CyclicBarrier** e **CountDownLatch** abstraem sincronização em alto nível. A CyclicBarrier permite sincronizar múltiplas threads repetidamente, útil para fases de algoritmos paralelos. Já o CountDownLatch permite que threads esperem até que um número de eventos ocorra. Estas ferramentas simplificam o código, reduzem erros e aumentam clareza no controlo de fluxo concorrente.

Questão 4 - Consistência e Relógios Lógicos

a) Como os relógios de Lamport ajudam a ordenar eventos em sistemas distribuídos?

b) O que permite que os relógios vetoriais identifiquem concorrência entre eventos?

c) Qual a utilidade dos relógios lógicos na replicação com consistência causal?

a) Os **relógios de Lamport** atribuem a cada evento um timestamp incremental. Ao enviar mensagens, o processo inclui o valor atual do seu relógio; o recetor ajusta o seu valor para o máximo entre o seu e o recebido, depois incrementa. Assim, se um evento *a* causa o evento *b*, então o timestamp de *a* será menor que o de *b*, permitindo uma **ordem parcial consistente com a causalidade**.

b) Relógios vetoriais mantêm um vetor com uma entrada por processo. Cada processo incrementa apenas a sua posição local ao gerar um evento, e funde vetores ao receber mensagens. Se dois vetores forem incomparáveis (nenhum é maior ou igual ao outro), então os eventos são **concorrentes**, isto é, sem relação causal. Esta precisão é impossível com relógios escalares.

c) Na **replicação com consistência causal**, os relógios vetoriais ajudam a garantir que uma operação só é aplicada numa réplica depois de todas as suas dependências causais terem sido aplicadas. Isso evita anomalias, como ver uma resposta antes de ver a pergunta, e permite replicação eficiente e consistente sem impor uma ordem total.