Cliente/Servidor:
No paradigma cliente/servidor, o servidor tem como função oferecer um serviço ao cliente, isto é, por exemplo um servidor web envia HTML da página pedida, servidor de mail entrega novos pedidos. O cliente efetua pedidos ao servidor e trata as mensagens, como por exemplo, cliente do browser mostra a página web recebida do servidor, normalmente também é o cliente que inicia contacto com o servidor. Em termos de comunicação, esta deve ser fiável, ou seja, não deve haver perda de dados e a entrega de mensagens deve ser ordenada e para que isto aconteça usamos canais TCP. De forma a estabelecer ligações, o servidor fica à espera de ligações numa determinada porta, já o cliente quando se liga ao servidor estabelece nova conexão bidirecional e os extremos da conexão representam sockets.

Explicar a operação wait(cond, lock):
As operações de wait(cond, lock) funcionam ao nível do sistema operativo com a ajuda do hardware. Estas soluções fazem parte do regime de exclusão mútua de forma a circundar secções críticas para escalonar o acesso das threads de um programa a essas regiões. Quando uma thread é a primeira a chegar a uma secção crítica que está rodeada por um lock (que aqui vou designa-lo por lock X) e adquire esse lock, todas as outras threads quando invocarem a instrução wait e um dos argumentos dessa instrução seja o lock X, essas threads vão ter de esperar até a thread que possui o lock X libertá-lo e alterar o valor boleano da condição que é passada como argumento na instrução wait. É também necessário invocar a instrução notifyAll para todas as threads, que estavam adormecidas, acordarem e continuarem a sua execução. Desta forma, percebemos que para poder invocar uma instrução wait, temos de ter sempre um lock associado.

Monitor é uma primitiva de alto nível:
O monitor é uma primitiva estruturada de controlo de maior nível de abstração. Este tipo de primitiva oferece um tipo abstrato de dados com controlo de concorrência implícita em todas as operações com exclusão mútua. Associa automaticamente os dados e operações ao código de controlo de concorrência. Disponibiliza variáveis de condição implícita ou explicitamente, o que é uma das suas vantagens. A principal desvantagem no uso deste tipo de primitiva é o facto de não haver tanto controlo sobre as secções críticas, por exemplo, em caso de fazermos um wait, no caso dos monitores, vamos fazê-lo a todos os processos o que pode ser desnecessário.

Diferenças ao assegurar exclusão mutua entre processos a executar numa única máquina ou num SD:
Em sistemas concorrentes podemos assegurar exclusão mutua em 2 tipos de sistemas que são o caso em que o sistema é um sistema distribuído ou o sistema é uma máquina única. No primeiro caso, isto é, um sistema com memória independente sendo que a exclusão mutua é obtida através do processo de troca de mensagens com primitivas simples do tipo send/receive, cliente-servidor, BroadCast/MultiCast, comunicações em grupo ou filas de mensagens. Estas comunicações podem ser síncronas ou assíncronas. No segundo caso a máquina é um sistema com memória partilhada, portanto utiliza exclusão mutua através de primitivas de sincronização, tais como locks, semáforos, variáveis de condição e monitores. Neste caso as ações dos processos são intercaladas de um modo imprevisível e o acesso concorrente a dados partilhados pode levar a inconsistência.

Funcionalidades de um servidor de objetos:
Um servidor de objetos ao contrário de um servidor tradicional não oferece um serviço específico, mas mantém um conjunto de objetos que o cliente use disponibilizando vários serviços. Um servidor de objetos tem como papel principal num sistema de objetos distribuídos, gerir um conjunto de objetos e intermediar os pedidos que são realizados aos objetos. Um servidor de objetos multi-thread podendo atribuir uma thread a cada objeto ou uma thread a cada invocação. Este tipo de servidor é constituído por servants, onde se situa a implementação dos objetos, a parte funcional. Os skeletons que são a proxy do lado do servidor, responsáveis por reestruturar as invocações e linearizar e enviar os resultados, e, por fim, pelo object adapter que é o gestor dos objetos oferecidos pelo servidor de objetos.

Algoritmo Centralizado: Um processo é escolhido para coordenar o acesso à zona crítica. Um processo que queira executar a sua zona crítica envia um pedido ao coordenador. O coordenador decide que processo pode entrar na zona crítica e envia a esse processo uma resposta. Quando recebe a resposta do coordenador, o processo inicia a execução da sua zona crítica. Quando termina a execução da sua zona crítica, o processo envia uma mensagem a libertar a zona.
Algoritmo Bully: O algoritmo é despoletado por um processo Pi que julga que o coordenador falhou. Pi envia uma mensagem de eleição a todos os processos com maior prioridade que Pi. Se num intervalo T Pi não receber qq resposta, então autoelege-se coordenador. Se receber alguma resposta então Pi espera durante T' por uma mensagem do novo coordenador. Se não receber nenhuma mensagem então reinicia o algoritmo. Se Pi não é o coordenador então, a qualquer momento pode receber uma mensagem...

Lock/unlock ao nível do SO:
As operações lock e unlock funcionam ao nível do sistema operativo com a ajuda do hardware. Estas soluções fazem parte do regime de exclusão mutua de forma a circundar secções críticas para escalonar o acesso das threads de um programa a essas mesmas regiões. Este tipo de primitivas de controlo de concorrência elimina esperas ativas. Quando uma thread adquire um lock, o kernel manda as outras threads “adormecerem”, isto é, coloca o estado delas a bloqueado até a thread que está a executar a secção crítica libertar o lock no fim, através de um unlock. Assim que o processo descrito libertar o lock (invocando o unlock), todos os outros processos que estavam adormecidos, passam do estado bloqueado para o estado pronto, sendo que ficam disponíveis para entrar na zona crítica se assim forem escalonados.

Descrever o algoritmo distribuído de exclusão mutua em anel:
O algoritmo distribuído de exclusão mutua em anel é aplicável a sistemas organizados física ou logicamente em anel. Neste algoritmo é assumido que os canais de comunicação são unidirecionais, cada processo mantém uma lista de ativos que consiste nas prioridades de todos os processos ativos quando este algoritmo terminar. Quando um processo Pi suspeita a falha do coordenador, Pi cria uma lista de ativos vazios, envia uma mensagem de eleição m(i) ao seu vizinho e insere i na lista de ativos. Por fim, se Pi recebe uma mensagem de eleição m(j) responde de uma de três formas: 1) se foi a primeira mensagem que viu, então cria uma lista de ativos com i e j e envia as mensagens de eleição m(i) e m(j), nesta ordem ao vizinho. 2) se i=!j então junta j à sua lista de ativos e reenvia a mensagem ao vizinho. 3) se i=j então a sua lista já contém todos os processos ativos no sistema e Pi pode determinar o coordenador.

Hibridas:
Superpeers: Organizar as redes peer-to-peer de forma hierárquica
Edge-Peers(CDN): Existe em qualquer operador de comunicação do mundo; Consiste em mandar um produto para os respetivos CDN's, para no dia seguinte esse produto ser enviado com rapidez para os users; Distribui a carga com eficiência e diminui os custos económicos
Bit-Torrent: Objetivo igual a edge-peers; Rede formada por todos os users, e não CDN's; Como funciona ? Temos um serviço centralizado, que nos dá a info de quem tem o quê e onde está. Assim podemos ir ver que users têm um certo conteúdo -> Trackers

Arquiteturas Descentralizadas de SD:
As arquiteturas descentralizadas apresentam-se como uma alternativa as arquiteturas centralizadas de sistemas distribuídos. A este tipo de arquitetura damos o nome de sistemas par-a-par ou peer-to-peer (P2P). Estes sistemas podem ser estruturas, nos casos em que obedecem a uma estrutura específica para a distribuição dos dados, ou podem ser não estruturados em que a interação entre pares é feita de forma aleatória. No primeiro caso, esta estrutura é obtida através do DHT (tabela de hash distribuída). No segundo caso, os sistemas não-estruturados funcionam de maneira mais aleatória, onde cada nó ligado à rede mantém uma vista parcial da mesma, consistindo num número razoavelmente pequeno de nós. Cada nó P escolhe periodicamente um nó Q da sua vista parcial. P e Q trocam informações e trocam nós das suas vistas parciais. A aleatoriedade com que as vistas parciais são mantidas é crucial para o funcionamento e robustez do sistema.

Das Arquiteturas estudadas qual se adequa melhor a um sistema de suporte de redes sociais:
Das arquiteturas estudadas, as que se adequam melhor a um sistema de suporte de redes sociais seriam a arquitetura de camadas e a baseada em eventos, embora a primeira opção seja a melhor para a situação em causa, pois oferece uma interface única e a capacidade de manter a persistência de dados. A arquitetura base em eventos era minimamente adequadas devido à comunicação através de eventos dos quais contém dados, isto é, publica-se um evento e o midleware garante que alguém tenha acesso ao evento. No entanto, a arquitetura em camadas é melhor neste caso.

Midleware orientado as mensagens:
O Midleware é uma camada de software que se estende por várias máquinas fornecendo uma abstração para a programação de aplicações e o seu papel é o de melhorar a transparência que um sistema distribuído deve ter, facilitar o acesso a recursos remotos/distribuídos, ter abertura e extensibilidade e escalabilidade. O midleware orientado às mensagens utiliza comunicação assíncrona, isto é, o emissor não sincroniza com o recetor e é persistente, ou seja, as mensagens são guardadas até serem entregues, através de filas de mensagens. Os sistemas de filas de mensagens assumem um protocolo de dados e da sua estrutura. No entanto, uma das principais aplicações dos sistemas de filas de mensagens é integrar diferentes aplicações desenvolvidas independentemente num sistema de informação distribuída coerente.

Algoritmo Descentralizado: Quando um processo Pi pretende aceder à sua zona crítica gera uma etiqueta temporal TS e envia um pedido (Pi, Tsi) a todos os processos. Quando um processo recebe um pedido pode responder logo ou adiar a sua resposta. Quando um processo recebe respostas de todos os processos no sistema pode então executar a sua zona crítica. Depois de terminar a execução da sua zona crítica, o processo responde a todos os pedidos aos quais adiou a resposta. A decisão de Pj responder logo a um pedido (Pi, TSi) ou adiar depende de três fatores: Se Pj estiver na sua zona crítica, adia. Se Pj não pretender aceder à sua zona crítica, responde. Se Pj pretender aceder à sua zona crítica (e já enviou também um pedido) então compara a etiqueta temporal do seu pedido TSj com TSi: Se TSj > TSi então responde logo (Pi pediu primeiro) Senão adia a resposta.
Algoritmo em anel: Aplicável a sistemas organizados física ou logicamente em anel. Assume que os canais de comunicação são unidirecionais. Cada processo mantem uma lista de ativos que consiste nas prioridades de todos os processos ativos quando este algoritmo terminar. Quando um processo Pi suspeita a falha do coordenador, Pi cria uma lista de activos vazia, envia uma mensagem de eleição m(i) ao seu vizinho e insere i na lista de ativos.

Baseada em Objetos: É usada quando não é eficaz usar por camadas, mas sim por um conjunto de entidades, que têm funções atómicas e que as vamos modelar e estabelecer ligações entre elas; Isto é feito á custa da noção de objetos. Usamos um application Server. Existe encapsulamento do estado dos objetos. Difícil de desenvolver.

Baseada em Dados: É usada quando a aplicação depende de um grande número de dados. O centro do sistema é uma grande infraestrutura partilhada onde temos dados recolhidos. A funcionalidade é através de componentes funcionais que consomem dados.

Diga o que entende por sincronização de processo em programação concorrente:
Em programação concorrente, os processos a executar são ações autónomas, a velocidade de cada ação é imprevisível, pois não existe maneira de saber quais as velocidades relativas. Dentro da programação concorrente existem 2 paradigmas: a comunicação e a sincronização. Neste caso, a sincronização pode estar ou não associado a comunicação, uma vez que os processos podem ter de esperar antes de prosseguir. Isto pode levar a esperas ativas, coisa que queremos evitar.

Processo: Dois processos não partilham a memória (mesmo que seja pai e filho), ou seja, são criados da mesma maneira, mas se a variável do pai é alternada, a do filho não irá mudar.Cada processo tem o seu espaço de endereçamento privado.
Thread: podem ser vistas como processos leves que permitem cooperação eficiente via memória; Podem também ser pedidos ao SO segmentos de memória partilhada entre processos;

```
public class Server {
private ServerSocket servsocket;
private int porto;
private Banco b;
public Server (int porto){
this.porto = porto;
this.b = new Banco();}
public void startServer(){
try {
this.servsocket = new ServerSocket(this.porto);
while(true){
Socket socket = servsocket.accept();
ServerWorker sw = new ServerWorker(socket, b);
new Thread(sw).start();}
} catch (IOException e) {
e.printStackTrace();}
public static void main(String[] args) {
Server s = new Server(12345);
s.startServer();}}

public class ServerWorker implements Runnable {
private Socket socket;
private Banco b;
public ServerWorker (Socket socket, Banco bank){
this.socket = socket; this.b = bank;}
public void run() {
try{
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
OpsBanco op;
while(true){
double saldo = 0; int id = 0; double valor = 0; op = (OpsBanco) in.readObject();
    try{
switch(op){
case CRIAR_CONTA:
System.out.println(OpsBanco.CRIAR_CONTA)
saldo = in.readDouble(); id = b.criarConta(saldo);
out.writeInt(id); out.flush(); break;
case FECHAR_CONTA:
System.out.println(OpsBanco.FECHAR_CONTA);
id = in.readInt(); saldo = b.fecharConta(id);
out.writeInt(OpsBanco.OK.ordinal()); out.writeDouble(saldo); out.flush(); break;
default:
out.writeInt(OpsBanco.OP_INVALIDA.ordinal());
out.flush();}
    } catch(ContaInvalida ci){
out.writeInt(OpsBanco.CONTA_INVALIDA.ordinal()); out.flush();}
}
} catch (IOException ioe) {
  try{
socket.shutdownOutput(); socket.shutdownInput(); socket.close();}
 catch(Exception e){
e.printStackTrace();    }
} catch (ClassNotFoundException ce){
ce.printStackTrace();  }
}

public class Client implements InterfaceBanco{
private Socket socket;
private ObjectOutputStream out;
```

```
private ObjectInputStream in;
public Client(String hostname, int porto){
try {
this.socket = new Socket(hostname, porto);
this.out = new ObjectOutputStream(socket.getOutputStream());
this.in = new ObjectInputStream(socket.getInputStream());
} catch (IOException e) {
e.printStackTrace();}}
public int criarConta(double saldo) {
int id = -1;
try{
//enviar id da operação e saldo da nova conta
out.writeObject(OpsBanco.CRIAR_CONTA);
out.writeDouble(saldo);
out.flush();
//receber id da nova conta
id = in.readInt();
} catch(IOException e){
e.printStackTrace();}
return id;}}
public static void main(String[] args) {
Client c1 = new Client("127.0.0.1",12345);
c1.criarConta(10);
System.out.println("Terminar clientes.");
c1.close();}}
```