Um **sistema [de operação] distribuído** é um sistema de operação cujos componentes, localizados nos diferentes elementos de um sistema computacional paralelo, comunicam e coordenam normalmente as suas acções por passagem de mensagens. A motivação principal que leva à construção e à utilização de sistemas distribuídos, é a partilha de recursos.
Dois modos como esta partilha se manifesta são a:
•paralelização de aplicações-tirando partido dos múltiplos processadores e outros componentes 'hardware' existentes no sistema computacional paralelo, procura-se garantir uma execução mais rápida e eficiente de diferentes aplicações;
•disponibilização de serviços-gerindo um conjunto de recursos, que estão de algum modo relacionados, procura-se fornecer de um modo consistente e seguro a sua funcionalidade aos diversos utilizadores e aplicações; tipicamente, o acesso aos recursos é controlado por um programa que oferece um interface de comunicação baseado num conjunto bem definido de operações.

A **concepção de sistemas distribuídos**, e das aplicações executadas sobre eles, envolve múltiplas questões que têm que ser ponderadas.
Entre elas destacam-se:
•a [eventual]heterogeneidade dos componentes que formam o sistema computacional paralelo;
•o grau de abertura que o sistema apresenta;
•a segurança dos fluxos de informação trocados e da informação armazenada;
•o potencial apresentado para aumentos de escala;
•o tratamento de falhas;
•a sincronizaçãoe a imposição de exclusão mútuano acesso a regiões críticas;
•o grau de transparência.

A **heterogeneidade** dos componentes que formam o sistema computacional paralelo, coloca-se em diversas vertentes:
•diferentes tipos de redes de computadores que estão interligadas;
•diferentes tipos de hardware apresentados pelos elementos computacionais do sistema paralelo;
•diferentes sistemas de operação em execução simultânea;
•diferentes linguagens de programação usadas na construção de aplicações;
•implementação de funcionalidades produzidas por diferentes fabricantes.
Face a esta realidade tão variada, é necessário encontrar soluções que mascarem a diversidade subjacente.

As soluções são em primeiro lugar localizadas, introduzindo um maior ou menor grau de uniformidade a um nível concreto da cadeia de comunicação:
•especificação e implementação de uma arquitectura de rede;
•transformação em representações comuns de especificidades próprias existentes na representação de dados em cada processador;
•harmonização do interface de programação de aplicações(API), no que respeita a comunicações, apresentado por diferentes sistemas de operação;
•adequação das estruturas de dados implementadas pelas diferentes linguagens de programação usadas na construção de aplicações;
•estabelecimento de standards aceites e obedecidos por diferentes fabricantes.

O grande objectivo, porém, é criar um modelo integrado de máquina virtual programável que mascare toda a heterogeneidade subjacente – middleware

O **grau de abertura** de um sistema computacional é uma característica que determina a sua maior ou menor capacidade para extensibilidade e reimplementação de maneiras diversas.
No caso de sistemas distribuídos, este grau de abertura tem fundamentalmente a ver com a capacidade de incorporação de novos serviços e da sua disponibilização a uma variedade alargada de aplicações.
Para que tal seja possível, torna-se necessário:
•estabelecer um mecanismo de comunicação uniforme no acesso a recursos partilhados;
•tornar públicos os seus interfaces principais;
•garantir a conformidade estrita, ao nível conceptual e de implementação, de cada novo componente com o standard estabelecido e tornado publico.

A disponibilização generalizada de serviços e a apetência demonstrada pelos sistemas computacionais paralelos para a execução cooperativa de aplicações levantam a questão de se saber quão seguros são os fluxos de informação trocados e quão segura é a informação armazenada nos recursos associados.
A segurança da informação pode ser vista a três níveis:
•confidencialidade-protecção contra a sua divulgação a entidades não autorizadas;
•integridade-protecção contra a sua alteração ou corrupção;
•disponibilidade-protecção contra interferências que perturbem os mecanismos de acesso

Algumas medidas podem ser tomadas para reduzir os riscos:

•introdução de firewalls-criando uma barreira, em torno de uma rede parcelar, que limita o tráfego de entrada e de saída a tipos bem definidos;
•encriptação das mensagens-mascarando o conteúdo dos fluxos de informação;
•utilização de assinaturas electrónicas-para certificação do autor da mensagem.
Há, contudo, situações de solução difícil:
•ataques conducentes a recusa de serviço -quando os fornecedores de um serviço são inundados por um número muito elevado de pedidos inúteis, que leva à sua paralização, impedindo na prática o acesso dos verdadeiros utentes;
•segurança do código móvel-ficheiros executáveis enviados como 'attachment' a mensagens, download de 'plug-ins' e 'applets', por exemplo, podem conduzir à realização de comportamentos indesejáveis, completamente fora docontrolo do utilizador local.

Os sistemas distribuídos devem permitir **aumentos de escala**. Isto quer dizer que, face a um crescimento significativo do número e/ou da dimensão dos recursos disponibilizados e/ou dos utilizadores que os solicitam, devem ainda assim manter um desempenho eficiente.
Desafios que se colocam:
•expansão dos recursos físicos-aumento de escala significa neste contexto que a quantidade de recursos necessários para servir nutilizadores é no máximo O(n);
•perda de desempenho -aumento de escala significa neste contexto que a perda de desempenho resultante do processamento de uma tarefa é no máximoO(log2n),em que né um parâmetro que mede de algum modo o tamanho da tarefa;
•evitar estrangulamentosfuturos -aumento de escala significa neste contexto que deve existir um planeamento eficaz da dimensão dos recursos e dos mecanismos de acesso que, face a um previsível crescimento futuro, impeça asua exaustão e perdas de desempenho.

Os sistemas distribuídos, sendo formados por múltiplos componentes hardware e software, são susceptíveis a **falhas parcelares de índole** extremamente variada. O seu tratamento é, por isso, muito difícil.
Alguns tipos de falhas são facilmente detectáveis, mas outros tipos são masca-rados pela complexidade do sistema. O grande desafio é como gerir o sistema num ambiente em que algumas falhas não podem ser detectadas, mas de cuja existência se pode meramente suspeitar.
Estratégias base para tratamento de falhas
•mascarar as falhas-algumas falhas que são detectadas, podem ser escondidas ou o seu efeito tornado menos severo
—as mensagens perdidas podem em muitos casos voltar a ser retransmitidas;
—a replicação de recursos pode permitir salvaguardar a informaçãoarmazenada quando há corrupção de dados em algun(s) deles.

Estratégias base **para tratamento de falhas**
•recuperar de falhas-para que tal seja possível, é necessário desenhar os componentes software de modo a que seja possível o armazenamento periódico do estado interno do processo; quando uma falha ocorre, o processamento é reiniciado a partir do último estado armazenado;
na prática, isto pode exigir a migração do código para outros recursos hardware e a utilização de réplicas actualizadas dos dados;
•tolerar as falhas-algumas falhas têm que ser toleradas; qualquer tentativa para as resolver é impossível, ou muito pouco prática; nestes casos, em vez de deixar o utilizador eternamente à espera, enquanto tentativas sucessivas de acesso são realizadas, é preferível alertá-lo para o facto.

Sendo a partilha de recursos a motivação principal que leva à concepção e implementação de sistemas distribuídos, é fundamental garantir que a entidade, ou entidades, que gere(m) o acesso a um recurso partilhado, impõe(m) um acesso com **exclusão mútua** para uma correcta operação.
Quando o controlo de acesso ao recurso é centralizado, dependente de uma única entidade, a solução é mais simples porque as primitivas de sincronização e de imposição de exclusão mútua desenvolvidas para ambientes de multiprogramação em monoprocessadores podem ser aplicadas.
Quando, porém, o controlo de acesso ao recurso é distribuído, dependente de várias entidades, a solução é muito mais complexa porque se coloca desde logo o problema de como sincronizar entidades não sujeitas a um relógio global.

O **grau de transparência de** um sistema computacional é uma característica que exprime o maior ou menor sucesso que foi conseguido a mascarar acomplexidade subjacente. A sua funcionalidade é descrita de uma forma integrada, conceptual-mente simples, em vez de resultar da interacção de um conjunto de componentes independentes.
O objectivo da transparênciaé, por isso, esconder os recursos que não são directamente relevantes para a tarefa em curso, tornando-os anónimos ao utilizador e/ou ao programador de aplicações.
Assim, o grau de transparência com que os diferentes recursos podem ser acedidos e manipulados, reflecte directamente o grau de abstracção e de operacionalidade apresentados pela camada middleware do sistema em causa.

Formas de transparência
•transparência no acesso-quando se efectuam o mesmo tipo de operações para acesso a recursos locais ou remotos;
•transparência de posição-quando se realiza o acesso aos recursos sem necessidade de conhecimento da sua localização precisa;
•transparência de rede-quando existe simultaneamente transparência no acesso e de posição;
•transparência de concorrência-quando o acesso a recursos partilhados se faz sem que ocorram interferências (a informação perma-nece consistente);
•transparência de replicação-quando é possível instanciar múltiplas réplicas dos recursos sem que tal se torne notório.
•transparência a falhas-quando as falhas, que ocorrem nos componentes hardware e/ou software, podem ser mascaradas e, por consequência, as tarefas em curso podem ser terminadas ;
•transparência de movimento-quando os utilizadores e os recursos podem ser movidos dentro do sistema, sem que tal afecte a realização das tarefas;
•transparência de desempenho-quando pode ocorrer uma reconfiguração dinâmica do sistema que leva em conta as variações de carga.

Proxy: quando uma invocação é feita a um objecto remoto, essa invocação é enviada de um modo transparente para um objecto local, que se chama proxy. A Proxy pode encaminhar o identificador da invocação e parâmetros, numa mensagem para o porto de um objecto remoto. Mas já que a proxy é um objecto, pode usar técnicas alternativas para implementar a invocação. Por exemplo, pode satisfazer a invocação com dados armazenados localmente em caches, ou pode fazer multicast da mensagem de invocação para uma colecção de objectos remotos.

Marshalling: é o processo que consiste em pegar numa colecção de dados e juntá-los (assembling) de uma maneira apropriada para a sua transmissão numa mensagem. Unmarshalling é o processo oposto, que consiste em separar os dados à sua chegada de modo a produzir uma colecção de dados equivalente no destino. Por outras palavras o marshalling consiste em nivelar as estruturas de dados numa sequencia básica de dados e traduzir esses dados numa representação de dados externa.

Modelo cliente - servidor:
Um dos modelos de sistemas distribuídos, é constituído por vários computadores pessoais (PCs e Workstations), um por utilizador, com os dados e as aplicações armazenados em servidores. Neste modelo, o utilizador é chamado de cliente e o sistema é chamado de modelo cliente servidor (client-server model). Neste modelo, a comunicação toma a forma de uma pedido do cliente ao servidor requisitando a execução de uma tarefa. O servidor executa a tarefa e envia os resultados ao cliente como resposta. Normalmente existem muitos clientes e um número pequeno de servidores. E, as aplicações são desenvolvidas de forma a serem atractivas e fáceis de utilizar, recorrendo para o efeito a interfaces gráficas. Este modelo está a substituir rapidamente os sistemas centralizados e outros modelos de sistemas distribuídos.

Existe um conjunto de características que distinguem o modelo cliente servidor de um sistema distribuído normal. Que são:
• Existe uma grande confiança em disponibilizar aplicações interactivas e atractivas, o que dá um grande controlo sobre a sua utilização;
• Apesar das aplicações estarem dispersas, existe uma enfâse na centralização das bases de dados e das aplicações. Isto permite às grandes empresas o controlo do investimento em sistemas computacionais e de informação e na disponibilização dos mesmos. E permite ao mesmo tempo aos departamentos individuais, escolher o tipo de máquinas e interfaces que necessitam para aceder à informação;
• Existe um compromisso das organizações e dos produtores para a construção de sistemas abertos e modulares. Permitindo ao comprador uma grande liberdade na escolha do equipamento e na sua interligação;
• As redes de comunicação são fundamentais, pelo que, a gestão e a segurança são aspectos prioritários.

A característica central numa arquitectura cliente servidor é a alocação das aplicações entre os clientes e os servidores. É bom não esquecer que o cliente e o servidor podem eventualmente usar sistemas operativos diferentes. Mas, isso é irrelevante enquanto o cliente e o servidor partilharem o mesmo protocolo de comunicação, por exemplo o TCP/IP, e suportarem as mesmas aplicações.

Nos servidores de ficheiros existe o problema da degradação no acesso a ficheiros remotos, relativamente ao acesso a ficheiros locais imposto pela rede de comunicações. Para aliviar este problema os clientes podem usar caches para manterem ficheiros requisitados recentemente ao servidor. Mas, num modelo cliente servidor em que vários clientes podem aceder e alterar ficheiros no servidor, existe o problema das cópias dos ficheiros mantidas nas caches locais dos clientes não estarem actualizadas (cache consistency).
A maneira mais simples de resolver o problema consiste em bloquear os ficheiros (file locking), para evitar o acesso simultâneo a um ficheiro. Um método mais eficaz praticado no Sprite, consiste em que, quando existe um processo de acesso para escrita num ficheiro aberto para leitura por outros processos, o servidor executa duas acções. O processo que abriu o ficheiro para escrita, pode manter uma cache mas, tem de escrever no disco do servidor, o

ficheiro logo após as alterações efectuadas. Em segundo lugar os processos que abriram o ficheiro para leitura são avisados que, o ficheiro já não é cacheable, ou seja, a versão mantida na cache local já não está em conformidade com a versão armazenada no servidor.

Passagem de mensagens

Normalmente, os sistemas computacionais constituintes de um sistema distribuído não possuem memória partilhada. Pelo que, as técnicas de comunicação de processos baseadas numa área comum de memória, por exemplo semáforos, não podem ser usadas. Em alternativa, a comunicação entre processos baseia-se na passagem de mensagens. Um processo cliente que precisa de um serviço envia uma mensagem ao servidor requisitando o serviço. O servidor aceita o serviço, respondendo com uma mensagem de resposta. Na sua forma mais simples, a passagem de mensagens é efectuada com apenas duas primitivas:

• A primitiva send especifica o destinatário e inclui a mensagem e enviar.

send (destination, &message);

• A primitiva receive especifica o remetente e fornece um buffer para armazenar a mensagem recebida.

receive (source, &message);

A forma das primitivas depende do software de passagem de mensagens. Pode ser a invocação de um procedimento, no caso do sistema operativo Unix é uma system call, ou uma mensagem para um processo que é parte do sistema operativo. A primitiva send é usada pelo processo que deseja enviar uma mensagem. Tem como parâmetros, a identificação do processo destinatário e o conteúdo da mensagem. O módulo de passagem de mensagens constrói uma estrutura de dados constituída por estes dois elementos. Esta estrutura de dados é enviada ao nó que alberga o processo destinatário através de um protocolo de comunicação, por exemplo, o TCP/IP. Quando a estrutura de dados é recebida pelo nó destinatário, é encaminhado para o seu módulo de passagem de mensagens. Este módulo identifica o processo a que se destina a mensagem e armazena a mensagem num buffer desse processo. O processo destinatário, tem de anunciar a intenção de receber a mensagem, através da primitiva receive e fornecendo o buffer.

As primitivas de passagem de mensagens podem ser do tipo bloqueante (síncrona) ou não bloqueante (assíncrona). Com primitivas de passagem de mensagens não bloqueantes, quando o processo remetente usa a primitiva send, o sistema operativo devolve o controlo ao processo, assim que terminar de copiar a mensagem ou de a colocar numa fila de espera para expedição. Quando o processo destinatário usa a primitiva receive, o sistema operativo devolve o controlo ao processo, e o processo é posteriormente informado da chegada da mensagem através de uma interrupção. Em alternativa, com primitivas de passagem de mensagens bloqueantes, quando o processo remetente usa a primitiva send, fica bloqueado até que a mensagem seja enviada, no caso de serviço não fiável, ou até que a mensagem seja enviada e seja recebida a confirmação da sua entrega, isto no caso de serviço fiável. Quando o processo destinatário usa a primitiva receive, fica bloqueado até que a mensagem seja colocada no buffer.

RPC

O modelo de passagem de mensagens baseia-se no paradigma de entrada/saída. Birrell e Nelson em 1984, propuseram uma variante deste modelo, baseado no paradigma procedural, que se designa por invocação de procedimentos remotos (Remote Procedure Calls). Este modelo permite a interacção de programas, em execução em diferentes nós de um sistema distribuído, através da invocação de procedimentos. A invocação de procedimentos remotos permite assim aceder a serviços remotos. Este modelo tem as seguintes vantagens:

• A invocação de um procedimento é uma abstracção muito usada;

• A invocação de um procedimento remoto, permite a especificação de operações bem definidas, com uma interface constituída pelo nome e tipos de dados associados, que pode ser divulgada, documentada e verificada;

• Como existe uma interface conhecida, o código de comunicação para uma aplicação pode ser gerado automaticamente;

• Uma especificação bem definida, permite ao programador desenvolver módulos cliente servidor que são portáveis com poucas modificações.

A invocação de um procedimento remoto, pode ser visto como um refinamento de um serviço fiável de passagem de mensagens bloqueantes. E deve ser ser o mais parecida possível, com a invocação de um procedimento local. A invocação de um procedimento remoto no programa cliente é ligada a uma bibioteca de procedimentos chamada client stub, que representa o procedimento do servidor no espaço de endereçagem do cliente. Por sua vez, o programa servidor está ligado a uma bibioteca de procedimentos chamada server stub.

Estas bibliotecas de procedimentos, stub do cliente e stub do servidor, escondem o facto de que a invocação do cliente ao servidor não é local. Os passos na invocação de um procedimento remoto, apresentados na figura do acetato anterior, são os seguintes:

1. O cliente invoca o stub do cliente, o que é uma invocação local e portanto os parâmetros são colocados na stack;
2. O stub do cliente tira os parâmetros da stack, faz o seu empacotamento numa mensagem (marshaling) e faz a invocação de uma system call para enviar a mensagem;
3. A kernel (do cliente) envia a mensagem do cliente para o servidor;
4. A kernel (do servidor) encaminha a mensagem para o stub do servidor;

5. O stub do servidor faz o desempacotamento dos parâmetros da mensagem, coloca os parâmetros na stack e invoca o procedimento do servidor.
A resposta percorre o caminho inverso, devolvendo ao programa do cliente os resultados produzidos pelo programa do servidor. Entretanto o cliente ficou bloqueado à espera da terminação do procedimento remoto.

Desta forma, em vez de se fazer uma entrada/saída através das primitivas send e receive, a comunicação remota é feita simulando a invocação de um procedimento. O acetato anterior, compara a invocação de um procedimento local com a invocação de um procedimento remoto. Apesar da elegância do conceito da invocação de um procedimento remoto, existem alguns problemas:
• Não é possível passar ponteiros para o procedimento, a não ser através de um truque. No caso de um ponteiro para uma variável, o stub do cliente pode empacotar a variável. O stub do servidor depois cria um ponteiro para essa variável e passa-o ao procedimento. No regresso a variável é devolvida e, caso tenha sido alterada, é copiada pelo stub do cliente substituindo o valor anterior. Infelizmente este truque não funciona para estruturas de dados complexas;
• Em linguagens como o C, em que é possível escrever procedimentos com parâmetros do tipo array sem especificar o seu tamanho, o stub do cliente não consegue empacotá-los pois não consegue determinar o tamanho efectivo do array;
• Procedimentos em que não é possível determinar o número e o tipo dos parâmetros, nem sequer através da especificação formal ou do próprio código, como por exemplo o printf, não podem ser invocados remotamente;
• Não é possível usar variáveis globais.
No entanto, a invocação de procedimentos remotos é largamente utilizada e na prática funciona bem, com algumas restrições. A invocação de procedimentos remotos pode ser síncrona ou assíncrona. A invocação síncrona é mais fácil de perceber e de programar, já que, o seu comportamento é previsível. No entanto, não explora totalmente o paralelismo inerente às aplicações distribuídas. Para permitir uma maior flexibilidade há implementações assíncronas. Uma utilização típica da invocação de um procedimento remoto assíncrono é a possibilidade de um cliente invocar um certo número de vezes seguidas o servidor, para conjuntos de valores diferentes, sem ter de esperar pelo resultado.