COMUNICAÇÃO ENTRE PROCESSOS

Comunicação entre processos

- Memória Compartilhada:
  - os processo compartilham variáveis e trocam informações através do uso dessas variáveis compartilhadas

- Sem Memória Compartilhada:
  - os processos compartilham informações através de troca de mensagens
  - o S.O é responsável pelo mecanismo de comunicação entre os processos

Comunicação entre processos - troca de mensagens

- Aspectos importantes para um sistema de troca de mensagens
  - Simplicidade: construção de novas aplicações para interoperar com já existentes deve ser facilitada
  - Semântica Uniforme: comunicação local (processos no mesmo nodo) e comunicação remota (processos em nodos diferentes) através de funções tão próximas quanto possível (=s !)
  - Eficiência: reduzir número de mensagens trocadas tanto quanto possível
    - economizar fechamento e abertura de conexões; usar piggybacking; etc.
  - Confiabilidade: garantir entrega da mensagem - confirmação, eliminação de duplicatas, ordenação
  - Corretude: relacionada principalmente a comunicação em grupo
    - garantia de aspectos como Atomicidade; Ordenação; “Survivability”

Comunicação entre processos - troca de mensagens

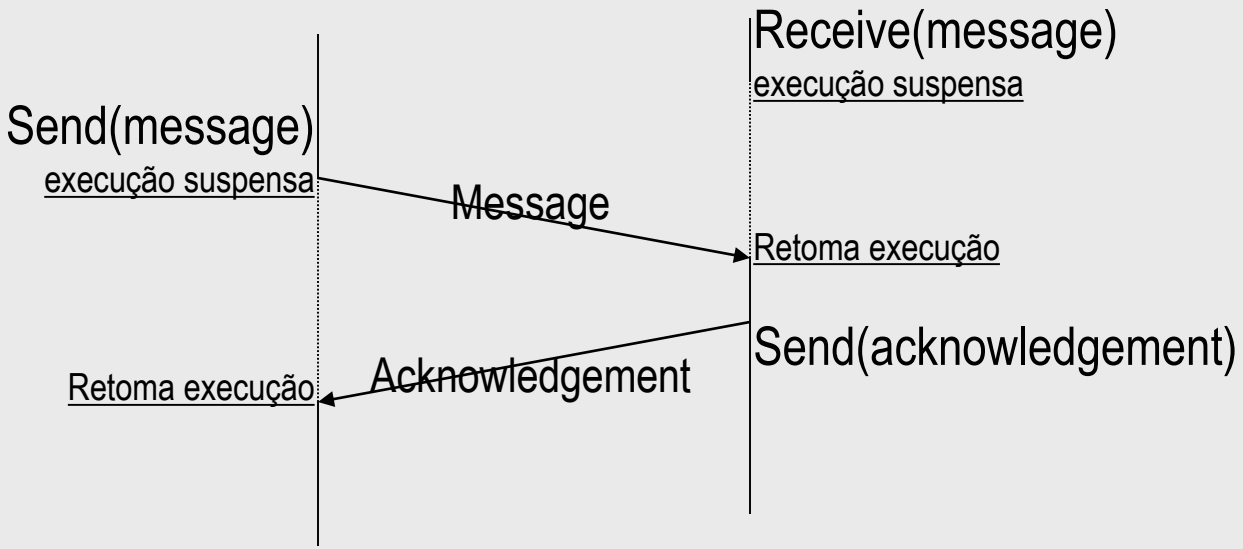
- Aspectos importantes para um sistema de troca de mensagens
  - Flexibilidade: possibilidade de utilizar somente funcionalidade requerida (em prol de desempenho)
    - necessidade ou não de entrega garantida, ordenada, atomica, etc
  - Segurança: suporte a autenticação, privacidade
  - Portabilidade: disponibilidade do mecanismo de IPC (*Inter Process Comunication*) em plataformas heterogêneas

Comunicação entre processos - troca de mensagens

- Operações: send/receive - características de sincronização: bloqueante, não bloqueante
  - Send bloqueante:
    - processo enviador fica bloqueado até recepção de confirmação do receptor
    - problema: ficar bloqueado para sempre - mecanismo de time-out
  - Send não-bloqueante:
    - processo enviador pode proceder assim que conteúdo (dados a enviar) for copiado para buffer de envio

Comunicação entre processos - troca de mensagens

- Operações: send/receive - características de sincronização: bloqueante, não bloqueante
  - quando send e receive são bloqueantes a comunicação é dita síncrona
  - senão é dita assíncrona



Comunicação entre processos - troca de mensagens

- Sincronismo é mais seguro, mas pode afetar desempenho
  - Ex: programação paralela
    - atrasando um processo (bloqueado até confirmação)
    - consumindo mais banda da rede que já é gargalo (mensagem de confirmação)
- Posso implementar diferentes níveis de bloqueio
  - Em buffer de saída do processo
  - Em buffer do ambiente de troca de mensagens
  - Em buffer do processo destino

Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: endereçamento implícito e explícito
  - explícito: processo com o qual se quer comunicar é explicitamente identificado através de um parâmetro
  - implícito: processo com o qual se quer comunicar não é identificado
    - ex.: sender mandando para qualquer receiver que desempenhe uma função/serviço - sender nomeia serviço ao invés de processo - qualquer processo servidor (de um grupo) que desempenhe esta função pode receber a mensagem
    - receiver quer receber independente do sender: modelo cliente/servidor - servidor quer poder servir qualquer cliente

Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: identificação de processos
  - machine_id@local_id - segunda parte usada localmente na máquina para achar processo
  - e se processo move ?
  - machine_id , local_id, machine_id
    nome de criação , máquina onde está
  - a máquina onde o processo foi criado deve manter tabela dizendo onde processo se encontra
  - máquinas que um processo visita tem que manter entrada dizendo a próxima máquina para onde o processo migrou; (ou só máquina origem?)
  - overhead; mensagem alcançar destino pode depender de vários nodos (falhas?)

Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: identificação de processos
  - métodos não transparentes: especifica identificador de máquina
  - identificador único do processo não deve ter embutida informação de localização do processo
  - Para melhor transparência: nomeação em dois níveis:
    - nome de alto nível (string) independente de localização
    - nome de baixo nível: como machine_id@local_id
    - *servidor de nomes* traduz de um para outro
    - processos usam nomes de alto nível para endereçar outros processos
    - durante o send o servidor de nomes é consultado para achar nome de baixo nível (localização)
    - uso possível para endereçamento funcional: originador diz nome de serviço e servidor de nomes mapeia para servidor apropriado

Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: bufferização
  - transmissão da mensagem: copiar corpo da mensagem do espaço de endereçamento do processo enviador para espaço de endereçamento do receptor
  - processo recebedor pode não estar pronto para receber
    - SO salva mensagem (SO do lado receptor implementa bufferização)
  - relação com o sincronismo da comunicação
    - síncrona - *null buffer* (um extremo)
    - assíncrona - *buffer* de capacidade ilimitada (outro extremo)
    - tipos intermediários de *buffers*: *single-message, finite-bound or multiple-message*

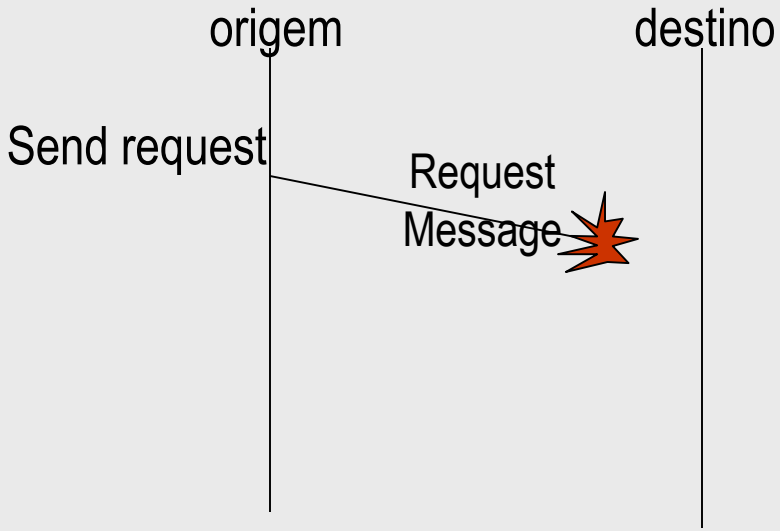
Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: codificação e decodificação dos dados da mensagem
  - mesagens entre processos rodando em diferentes arquiteturas - big-endian
  - transferência de valores de ponteiros para memória perdem significado
  - identificação necessária para dizer tipo de dado sendo transmitido

  - uso de formato comum de transferência (sintaxe de transferência)
    - representação com tags (tagged): mensagem carrega tipo dos dados transferidos. Ex.: ASN.1 (abstract syntax notation - CCITT); sistema operacional Mach e MPI (MPI data types: MPI_INT)
    - representação sem tags: receptor tem que saber decodificar mensagem. Ex.: XDR (eXternal Data Representation - Sun); Courier (Xerox)

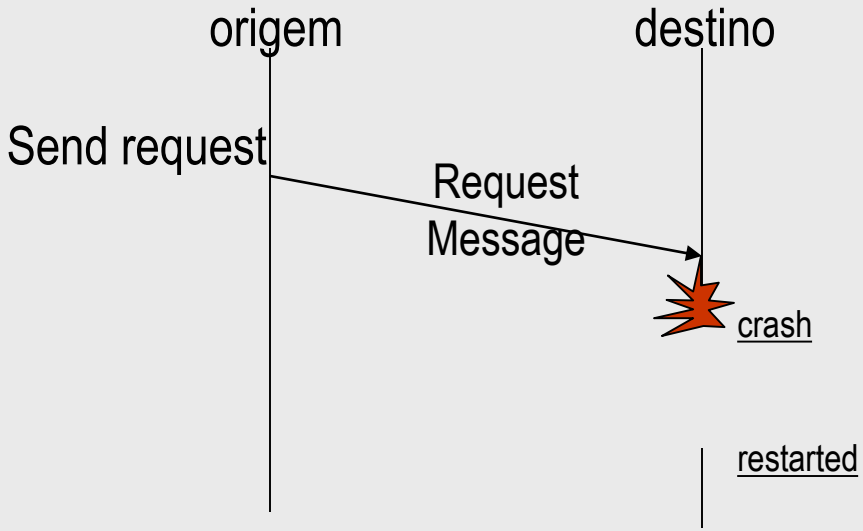
Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: tratamento de falhas
  - perda de mensagem de pedido



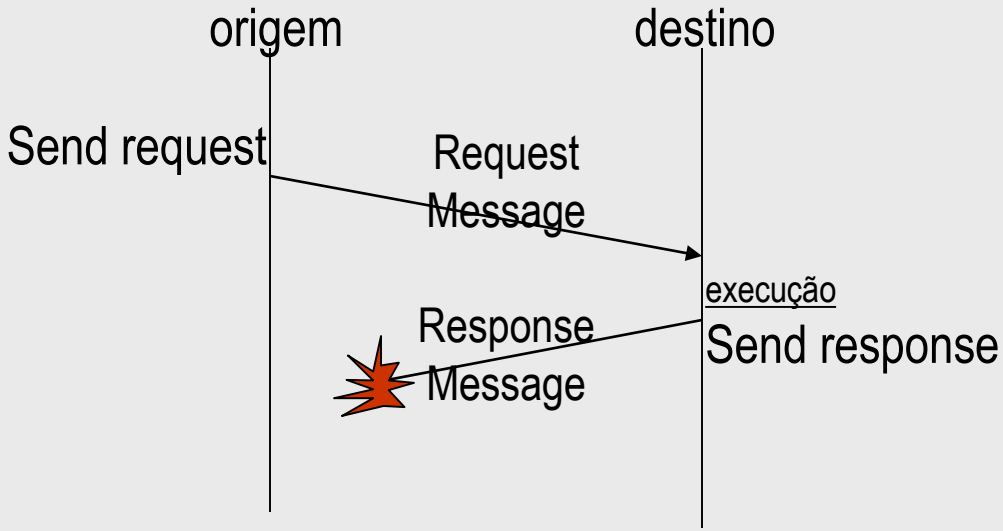
Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: tratamento de falhas
  - execução do pedido no destinatário não tem sucesso



Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: tratamento de falhas
  - perda de mensagem de resposta



Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: tratamento de falhas
  - estratégias
    - four message: confirmação das mensagens de pedido e resposta
    - three message: confirmação da resposta
    - two message: resposta é confirmação

Comunicação entre processos - troca de mensagens

- Operações: send/receive - característica: tratamento de falhas
  - idempotência e tratamento de requests duplicados
    - idempotência: repetibilidade
      - operação idempotente pode ser repetida inúmeras vezes sem causar efeitos colaterais ao servidor
        - ex.: get-time(), sqrt(x) OK
        - ex.: conta.deposita(valor); conta.retira(valor) ERRO!
    - operações não idempotentes
      - necessitam “exactely-once” semantics
      - garante que somente uma execução no servidor é realizada
      - ex.: usar identificador único para cada request; lado servidor guarda reply cache para responder mesma resposta a pedido repetido

Comunicação entre processos - troca de mensagens

- Troca de mensagens: Mecanismos
  - Arquivos (Unix *pipes*)
  - *Sockets*
  - RPC (*Remote Procedure* Call)
    - RMI (Remote Method Invocation) - Java, DCOM, CORBA, …
  - MPI

Comunicação entre processos - troca de mensagens

- Usando Unix *pipes*

```
int main ( ) {
   int b [ 2 ] ;  int p[2];  char m[128];

   p = pipe();    /** cria o pipe para comunicação **/
   id = fork () ;  /** cria outro processo **/
   if ( id != 0) {  /** código do processo pai **/
      write ( p[0],"message" );
    }
   else {          /** código do processo filho **/
        read( p[1], m );
        print ("%s\n",m );
    }
}
```

Comunicação entre processos - sockets

Server

```
int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n = read(newsockfd,buffer,255);
     if (n < 0) error("ERROR reading from socket");
     printf("Here is the message: %s\n",buffer);
     n = write(newsockfd,"I got your message",18);
     if (n < 0) error("ERROR writing to socket");
     return 0;
}
```

Client

```
int main(int argc, char *argv[])
{
    int sockfd, portno, n;

    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr
*)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
         error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```

Comunicação entre processos - RPC

Server Side

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

void *rusers();

main()
{
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
        RUSERSPROC_NUM, rusers,
        xdr_void, xdr_u_long,
        "visible") == -1) {
           fprintf(stderr, "Couldn't Register\n");
            exit(1);
          }
   svc_run(); /* Never returns */
  fprintf(stderr, "Error: svc_run returned!\n");
  exit(1);
}
```

Client Side

```
#include <stdio.h>
#include <utmp.h>
  ıclude <rpc/rpc.h>
  ıclude <rpcsvc/rusers.h>

   a program that calls the RUSERSPROG
  RPC program


  .n(int argc, char **argv)


   unsigned long nusers;
   enum clnt_stat cs;
   if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
    }

   if( cs = rpc_call(argv[1], RUSERSPROG,
          RUSERSVERS, RUSERSPROC_NUM, xdr_void,
          (char *)0, xdr_u_long, (char *)&nusers,
          "visible") != RPC_SUCCESS ) {
              clnt_perrno(cs);
              exit(1);
            }

   fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
   exit(0);
}
```

Comunicação entre processos - MPI

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv)
{
int my_rank;  /* Identificador do processo */
int proc_n;   /* Número de processos */
int source;   /* Identificador do proc.origem */
int dest;     /* Identificador do proc. destino */
int tag = 50; /* Tag para as mensagens */

char message[100]; /* Buffer para as mensagens */
MPI_Status status; /* Status de retorno */

MPI_Init (&argc , & argv);

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

if (my_rank != 0)
   {
   sprintf(message, "Greetings from process %d!", my_rank);
   dest = 0;
   MPI_Send (message, strlen(message)+1, MPI_CHAR,dest, tag, MPI_COMM_WORLD);
   }
else
   {
   for (source = 1; source < proc_n; source++)
      {
      MPI_Recv (message, 100, MPI_CHAR , source, tag, MPI_COMM_WORLD, &status);
      printf("%s\n", message);
      }
   }
MPI_Finalize();
}
```