Componentes e Especificação de Sistemas Tempo Real

O que é um Sistema Tempo Real?

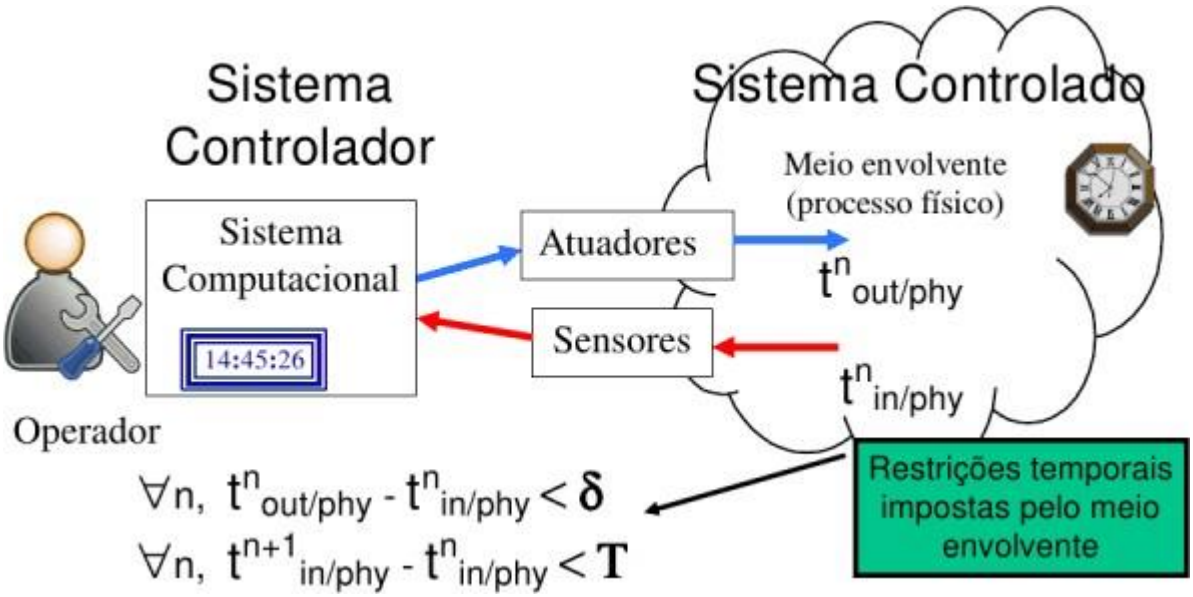
Existe uma grande diversidade de definições relacionadas com Tempo Real, os sistemas que lidam com Tempo Real, os serviços que prestam e as funcionalidades que desempenham. Em comum há o fato de dependência de um sistema computacional face ao tempo tal como existe num determinado processo físico.

A funcionalidade ou serviço de tempo real tem de ser desempenhada ou prestada dentro de intervalos de tempo finitos impostos por um processo físico. Os sistemas tempo real (STR) são
aqueles que desempenham pelo menos uma funcionalidade de tempo real ou que prestam pelo menos um serviço de tempo real.

A ciência de tempo real é o ramo da ciência da computação que estuda a introdução de tempo real nos sistemas computacionais. O tempo real, como propriedade de um sistema computacional, é caracterizada pela capacidade de um sistema computacional de estabelecer correspondências entre diferentes sistemas de medição e/ou contagem de tempo.

Assim nos sistemas operacionais tempo real os resultados das computações devem ser: logicamente corretos e produzidos a tempo. A figura abaixo ilustra esta relação.

A figura abaixo ilustra um diagrama de operação de um STR controlando um processo.



Nota-se que o meio envolvente com o qual o sistema computacional interage (processo físico) possui o seu próprio "ritmo de evolução", isto é, a sua própria dinâmica. Esse ritmo é inerente ao próprio processo físico e não pode (ou não deve, caso dos simuladores) ser controlado externamente. Designa-se, assim, o respeito ao ritmo, de tempo real.

Logo, o meio envolvente impõe ao sistema requisitos temporais de acordo com o seu tempo real (dinâmica). Para que o sistema computacional seja capaz de interagir com o seu meio envolvente, tem de atuar sobre ele a tempo, isto é, de acordo com o respectivo tempo real.

Deve-se atentar que tempo real não significa rapidez, mas apenas um ritmo de evolução próprio de um certo processo físico, ou seja, determinismo temporal. Por exemplo: um piloto de F1 (mas aplica-se a qualquer condutor ou a um robô...). O controle do volante tem de ser preciso, qualquer que seja a velocidade a que o carro circula.
Todos os eventos inesperados que surjam enquanto o carro é pilotado devem ser tratados a velocidade atual. A velocidade do carro determina o tempo real. Não é possível parar instantaneamente para pensar!

Genericamente, quando um sistema de controle ou monitoração consegue acompanhar o estado de um dado processo físico e, se necessário, atuar a tempo sobre ele, então trata-se de um sistema de tempo real.

Quando construímos máquinas (programáveis) para interagir com processos físicos, necessitamos usar técnicas de programação e infraestrutura de *software* que nos permita ter confiança na capacidade de atuação pontual.

Objetivos de Estudo dos SRT

O principal objetivo do estudo dos Sistemas Tempo Real (STR) é o de desenvolver técnicas de projeto, análise e verificação que permitam obter garantias de que um dado sistema, que se pretende de tempo real, possibilitando que tenha comportamento temporal adequado à dinâmica do sistema com o qual deve interagir.

Relativamente às atividades computacionais dos STR, os aspectos que normalmente mais interessam caracterizar são:

- tempo de execução tempo de
- resposta
- regularidade de eventos periódicos

Alguns aspectos particularmente importantes relativamente ao: Tempo

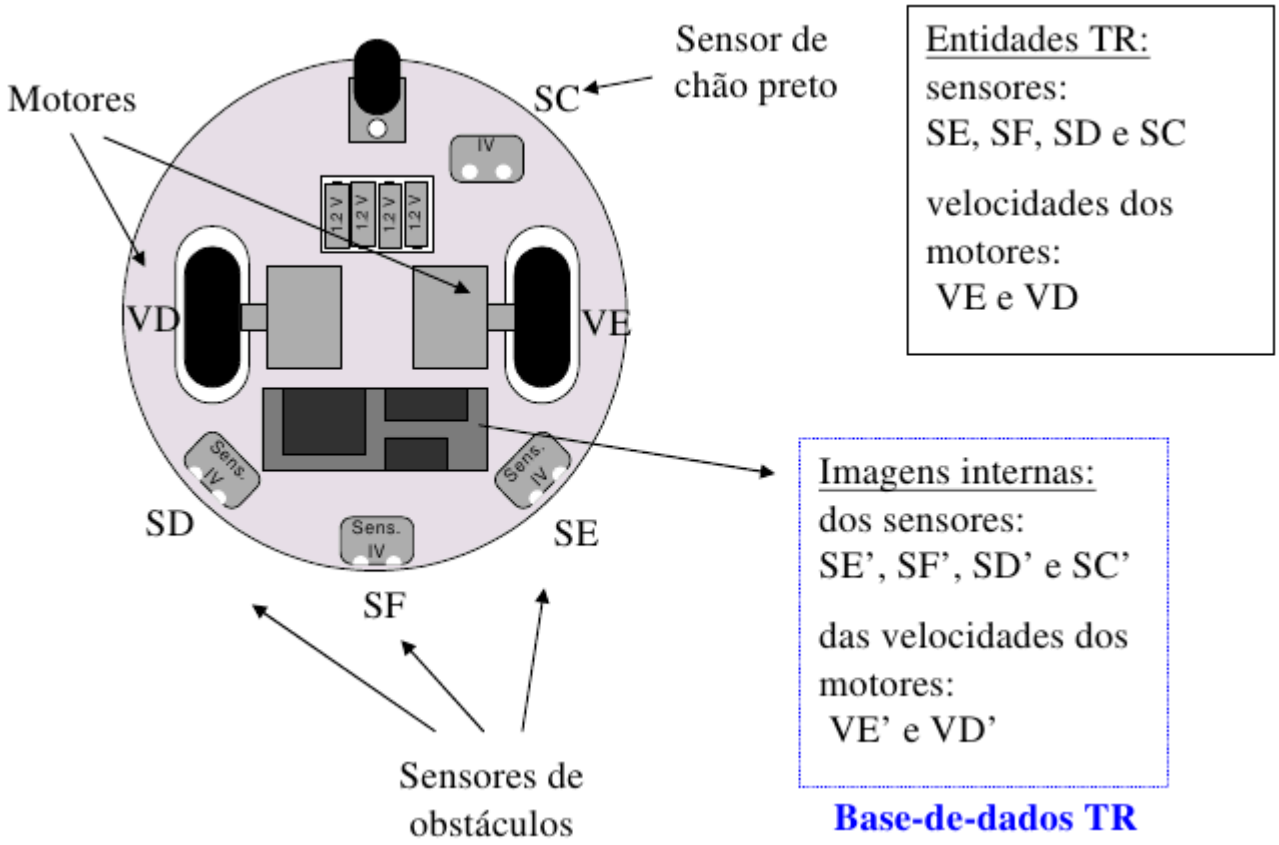
- de execução
  - Estrutura do código (linguagem, condicionais, ciclos) DMA,
  - *caches*, *pipeline*
  - Sistema Operacional ou *kernel* (*system calls*) Tempo
- de resposta e regularidade Interrupções
  - *Multi-tasking*
  - Acesso a recursos partilhados (barramentos, portas de comunicação, I/O, etc.)

Requisitos dos STR

Os requisitos tipicamente impostos aos Sistemas de Tempo Real são de 3 tipos: Funcionais

- Coleta de dados (*Data Sampling*) - Amostragem das variáveis do sistema (entidades de tempo real) quer do tipo contínuo quer discreto
- Controle Digital Direto (DDC) - Acesso direto do sistema controlador aos sensores e atuadores Interação com o operador (MMI ou HMI) - Informação do estado do sistema, registo histórico, suporte à correta operação do sistema
- Temporais
- de Dependabilidade

Na coleta de dados, internamente ao sistema controlador existem “imagens” locais (variáveis internas) das entidades de tempo real do sistema. Cada imagem de uma entidade de tempo real tem uma validade temporal limitada devido à dinâmica do processo físico. O conjunto das imagens das entidades de tempo real compõe a base de dados de tempo real. A base de dados de tempo real tem que ser atualizada sempre que houver uma mudança de valor numa entidade de tempo real. Por exemplo: um robo móvel com o o da figura abaixo.

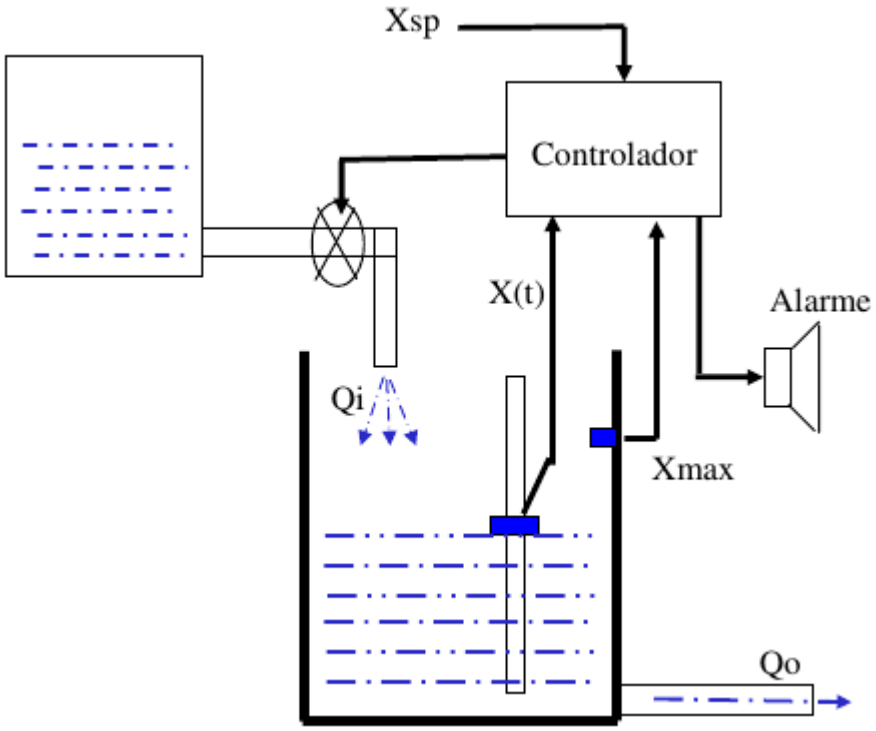


Os requisitos temporais normalmente advêm da dinâmica do processo físico que se pretende controlar. Por isso restrições são impostas, como:

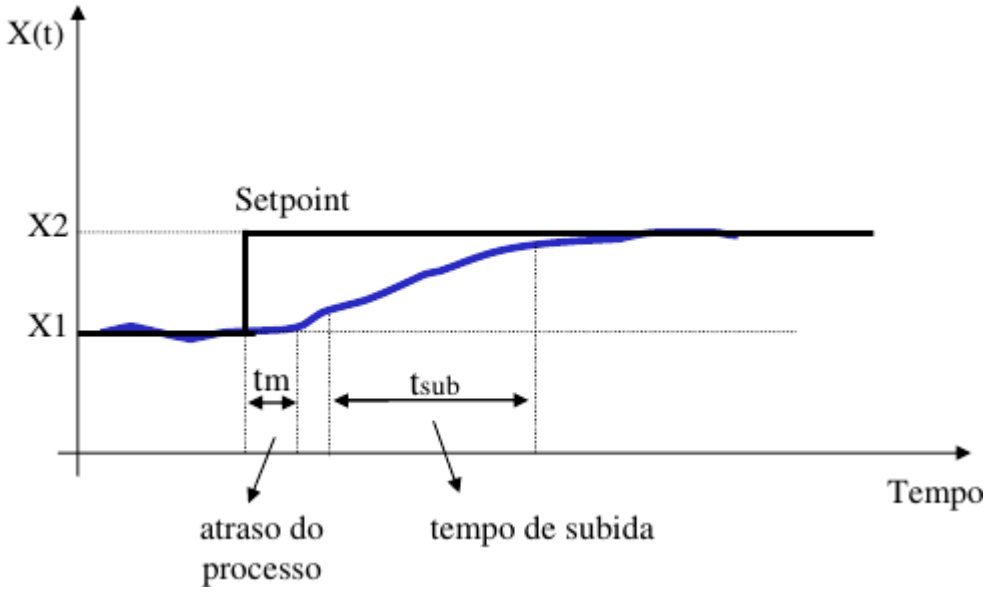
- atrasos de observação do estado do sistema
- atrasos de computação dos novos valores de controle (atuação) variações dos
- atrasos anteriores (*jitter*)

Estas restrições têm de ser cumpridas em todas as instâncias (incluindo o pior caso) e não apenas em termos médios.

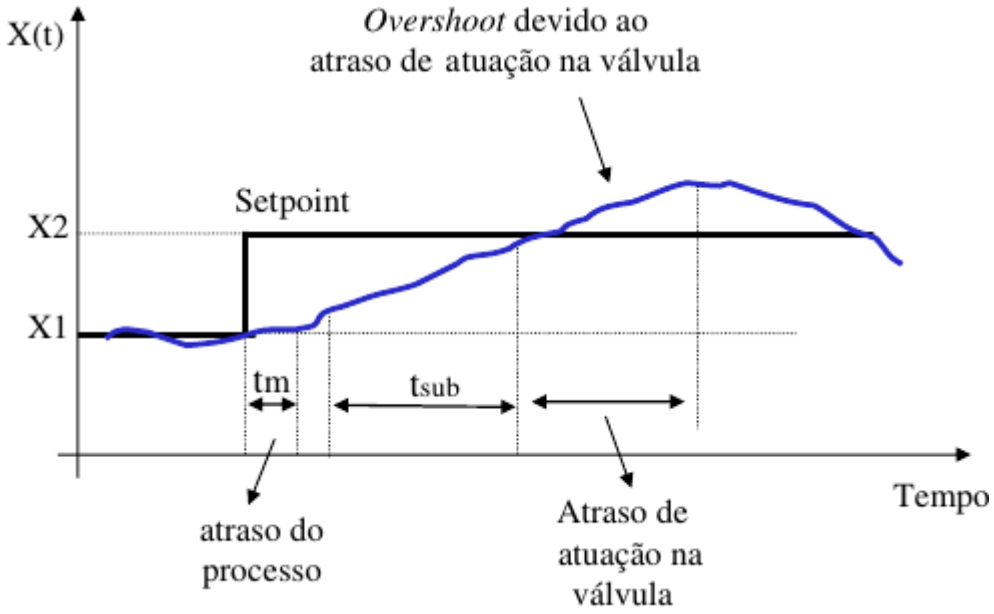
Exemplo: Controle de nível de líquido mostrado na figura abaixo.



Para se efetuar o controle o controlador atua na válvula com base nos sensores de nível (X(t)) e de nível máximo. A atuação do controlador pode ser descrita pelas métricas apresentadas no gráfico abaixo.



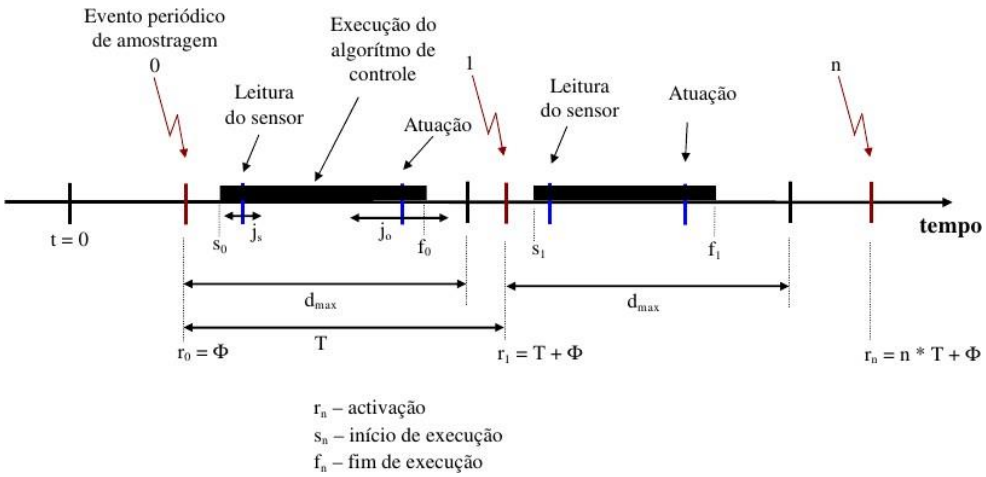
Nota-se a presença de atraso e um tempo de subida relativo ao controle do processo. Caso haja um atraso também na atuação podemos ter um controle degradado por atraso como mostrado na figura abaixo, onde o overshoot aparece evidenciando problemas no controle.



Para controlar sistemas deve-se obedecer alguns critérios em um STR, como: período de

- amostragem – Ts (critério de Nyquist)
- Atraso máximo na resposta da válvula dmax deve ser menor que Ts → facilmente compensado Variações no
- atraso de leitura do nível (*jitter*) deve ser muito menor que dmax
- Variações no atraso de atuação da válvula (*jitter*) deve ser muito menor que dmax → difíceis de compensar Atraso
- máximo na sinalização de alarme – dal,max

Assim, olhando para a execução do sistema de controle no STR teríamos a seguinte sequência de processamento.



Há portanto, uma classificação das restrições temporais (de acordo com a utilidade do resultado para a aplicação):

- Suave (*Soft*) – Restrição temporal em que o resultado que a ela está associado mantém alguma utilidade para a aplicação mesmo depois de um limite D embora haja uma degradação da qualidade de serviço.
- Firme (*Firm*) – Restrição temporal em que o resultado que a ela está associado perde qualquer utilidade para a aplicação depois de um limite D.
- Rígida (*Hard*) – Restrição temporal que, quando não cumprida, pode originar uma falha "catastrófica".

De acordo com o tipo de restrição temporal há também uma classificação das restrições temporais:

- ***Soft Real-Time*** – O sistema apenas apresenta restrições temporais do tipo *firm* ou *soft* (ex.: simuladores, sistemas multimedia)
- ***Hard Real-Time*** – O sistema apresenta pelo menos uma restrição temporal do tipo hard. São sistemas de segurança crítica (ex.: controlo de vôo de aviões, de mísseis, de centrais nucleares, de fábricas de produtos perigosos)

Já nops requisitos de dependabilidade há diversos aspectos importantes num sistema de segurança crítica, como:

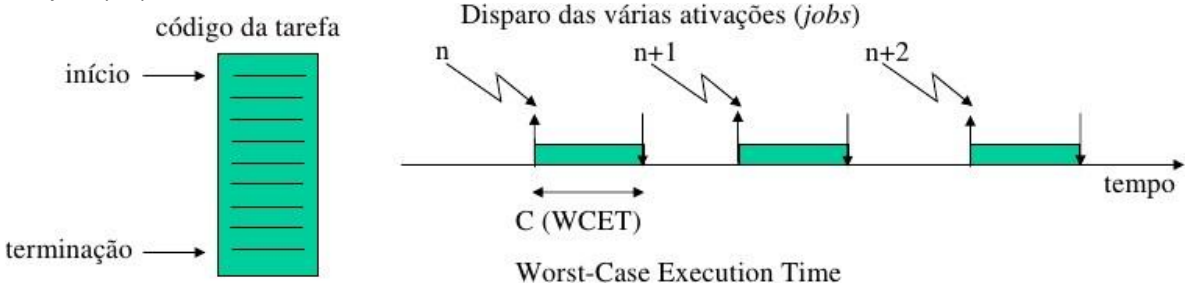
- Interfaces estáveis entre os subsistemas críticos e os restantes por forma a evitar a propagação de erros de uns para os outros.
- Cenários de pior caso bem definidos. O sistema deve possuir os recursos adequados para fazer face aos cenários de pior caso sem necessidade de recurso a argumentos probabilísticos, isto é, deve fornecer garantias de serviço mesmo em tais cenários.
- Arquitetura composta por subsistemas autônomos, cujas propriedades podem ser verificadas *independentemente* uns dos outros (composabilidade).

Modelos Computacionais

Para que se possa definir os componentes de um STR é importante vermos os modelos computacionais existentes. Há 3 tipos de modelos computacionais:

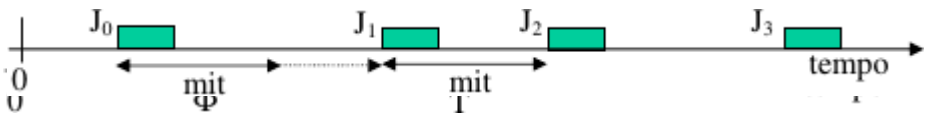
- Modelo transformacional -no qual um programa inicia e termina, transformando dados de entrada em resultados ou dados de saída.
- Modelo reativo - no qual um programa pode executar indefinidamente uma sequência de interações, por exemplo operando sobre um fluxo de dados.
- Modelo de tempo real -Modelo reativo em que o programa tem de se manter sincronizado com o fluxo de dados, o qual impõe restrições temporais à execução do programa.

Já vimos um pouco sobre processos e a definição deles é muito importante para se entender os conceitos de STR. Uma tarefa (intância de processo, ou atividade) é uma sequência de ativações (instâncias ou jobs). Cada uma composta por um conjunto de instruções que, na ausência de outras atividades, é executada pelo CPU sem interrupção. A figura abaixo ilustra o dsparo de vários jobs que podem formar uma tarefa.



Quanto à periodicidade as tarefas podem ser: Periódicas - instância n ativada em $a_n = n*T + \Phi$

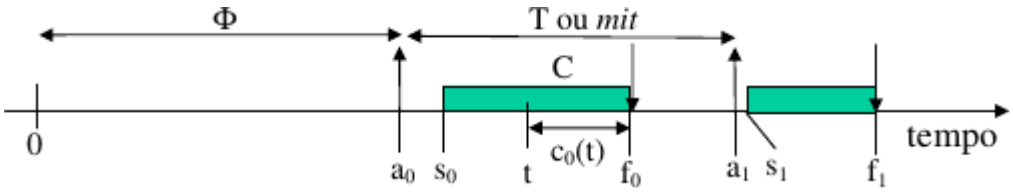
- Esporádicas - tempo mínimo entre ativações consecutivas (mit)



- Aperiódicas - só se caracterizam de forma probabilística

As tarefas possuem parâmetros que as caracterizam, como: C –

- tempo máximo de execução (WCET)
- T – período (para periódicas)
- Φ – fase relativa = instante da 1a ativação (para periódicas) mit –
- *minimum interarrival time* (para esporádicas)
- a_n – instante de ativação da na instância
- s_n – instante de início de execução da na instância f_n –
- instante de terminação da n instância
- $c_{n(t)}$ – tempo máximo de execução residual da na instância no instante t



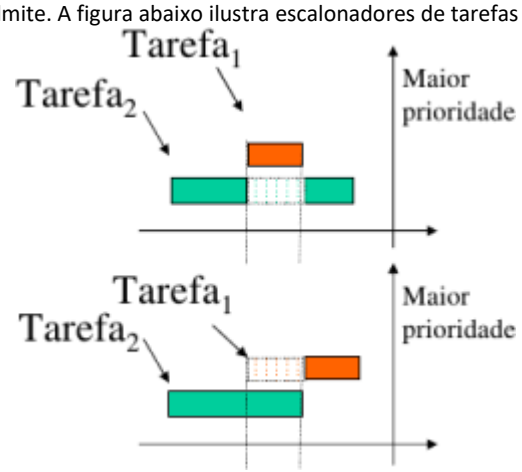
Os requisitos das tarefas podem ser:

- Temporais – limites temporais aos instantes de terminação ou de geração de determinados eventos de saída. Precedência –
- estabelecem uma determinada ordem de execução entre tarefas.
- Uso de recursos – necessidade de utilização de recursos partilhados (ex. portas de comunicação, um bu"er em memória partilhada, variáveis globais, periféricos do sistema). Pode implicar uso de operações atômicas (cuja sequência não pode ser interrompida).

Assim é importante termos em mente o conceito de preempção, que é:

- Quando uma tarefa pode ser interrompida temporariamente para execução de outra mais prioritária, diz-se que admite preempção.
- Quando um sistema utiliza a propriedade de preempção das tarefas que executa diz-se preemptivo.

Um conjunto de tarefas diz-se admitir preempção total quando todas as tarefas admitem preempção em qualquer ponto da sua execução (tarefas independentes). Assim, o acesso a recursos partilhados (tarefas com dependências) pode impor restrições sobre o grau de preempção que uma tarefa admite. A figura abaixo ilustra escalonadores de tarefas com preempção e sem preempção.

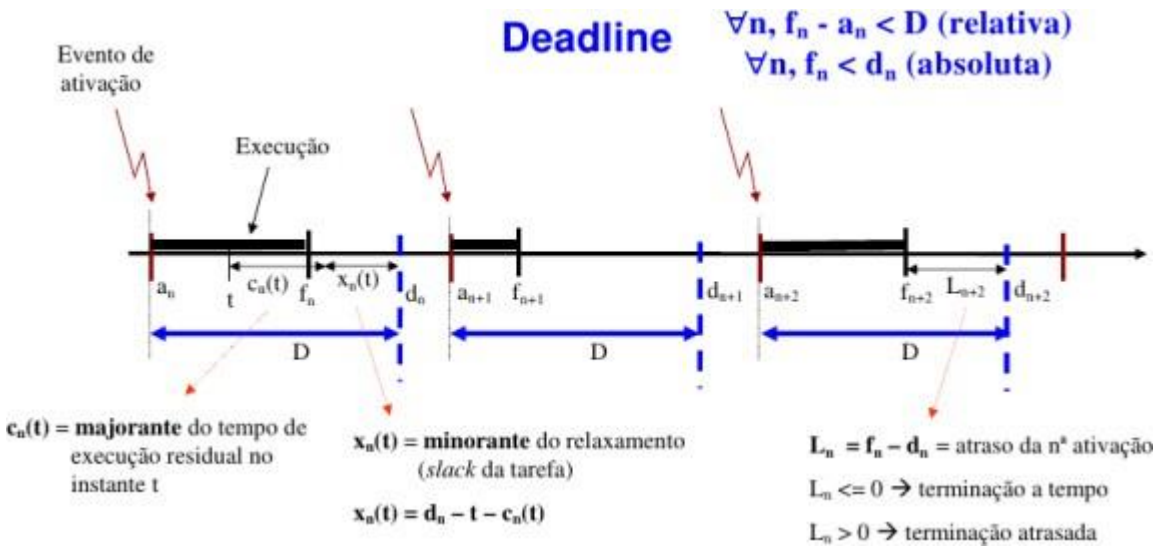


Nota-se no primeiro gráfico que escalonadores com preempção, quando surge uma tarefa (tarefa 1) com maior prioridade que a corrente (tarefa 2) esta é interrompida até que a com maior prioridade seja finalizada. Já nos escalonoadores sem preempção (segundo gráfico) a tarefa com menor prioridade não é interrompida mesmo que uma com maior prioridade "surja".

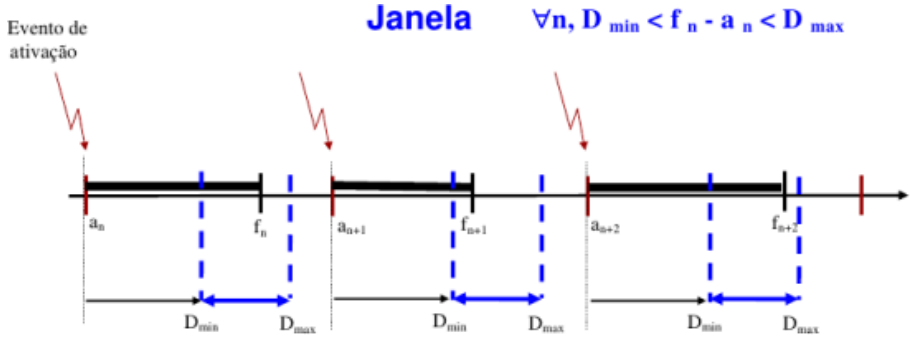
Os requisitos temporais das tarefas podem ser de vários tipos:

- *Deadline* – Limitação ao tempo máximo para terminação da tarefa, ou seja, um prazo Janela –
- Delimitação máxima e mínima ao instante de terminação.
- Sincronismo – Limitação à diferença temporal entre a geração de dois eventos de saídas.
- Distância – Limitação ao atraso (distância) entre a terminação, ou ativação, de duas instâncias consecutivas (ex. a mudança do óleo do motor num carro)

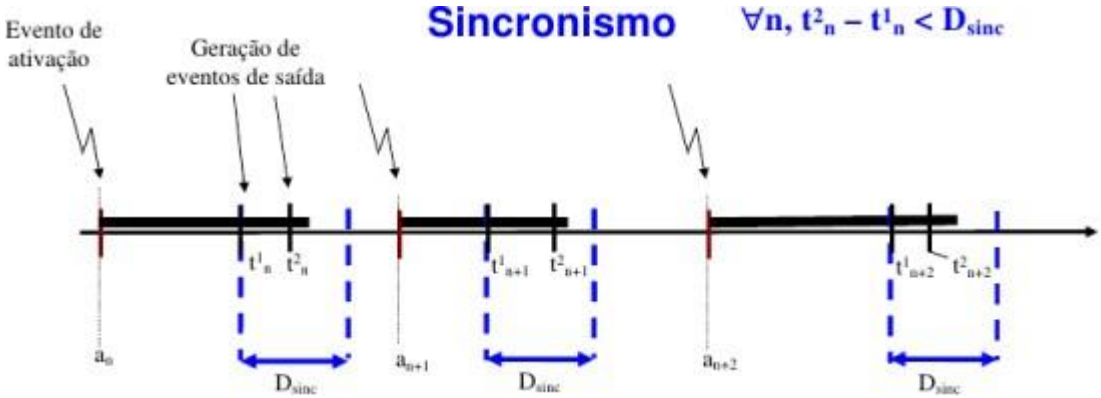
Destes requisitos o tipo *deadline* é de longe o mais comum em STR. A figura abaixo ilustra melhor um requisito temporal de *deadline* imposto a tarefa.



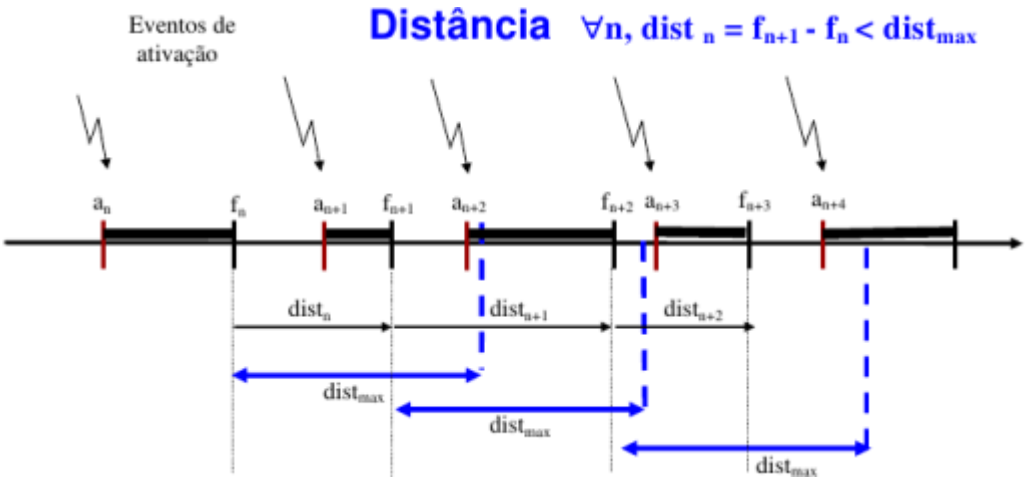
Nota-se que com relação ao cumprimento do(s) *deadline*(s) há vários parâmetros que devem ser atendidos. Com relação ao requisito temporal do tipo janela a figura abaixo ilustra os parâmetros relacionados.



Com relação ao requisito temporal do tipo sincronismo a figura abaixo ilustra os parâmetros relacionados.



Por último com relação ao requisito temporal do tipo distância a figura abaixo ilustra os parâmetros relacionados.



Implementação de aplicações de tempo real

A programação de aplicações de tempo real quando envolve apenas um ciclo principal e, eventualmente, um número muito reduzido de atividades assíncronas (que podem ser encapsuladas em rotinas de interrupção) é normalmente efetuada de forma direta sobre o CPU, isto é, sem recurso a estruturas de software intermediárias como SO ou *kernel*.

Por outro lado, quando a aplicação envolve múltiplas atividades, assíncronas ou não, a respectiva programação é facilitada pela utilização de sistemas operacionais multitarefa (*muti-tasking*) os quais suportam diretamente múltiplas tarefas que podem executar de forma independente, ou partilhando recursos do sistema. Neste caso cada atividade é encapsulada numa tarefa.

Em SOs multitarefas a programação de aplicações com recurso a estruturas de *software* (SO) permite: Maior nível de

- abstração
- Menor dependência relativamente ao *hardware*
- Maior facilidade de manutenção do *software*

O processamento associado a uma dada atividade pode ser efetuado:

- A nível de uma ISR (*Interrupt Service Routine*) onde não se tira partido de algumas vantagens do SO (programação de baixo nível – muito dependente do *hardware*) e há uma elevada reatividade a eventos externos (microssegundos). Também há grande interferência ao nível das tarefas e os SOs possuem um número limitado de ISRs.
- A nível de uma tarefa onde se tira partido das vantagens do SO (programação de alto nível, menor dependência do *hardware*, melhor manutenção) e há menor reatividade a eventos externos (maior *overhead*). Também há ISRs reduzidas para menor perturbação sobre as tarefas.

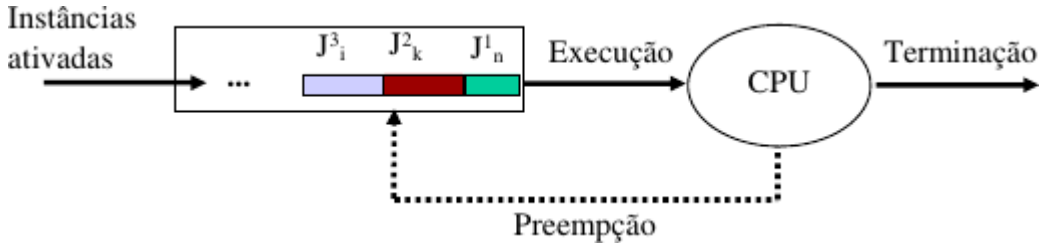
Os SOs multitarefa podem ser classificados relativamente às garantias temporais da seguinte forma:

- Não Tempo Real (*time-sharing*) - *Unix*, *Linux*, *Windows NT* - seguem modelo transformacional onde não é possível determinar o tempo de resposta a um evento (ex.: devido a *swapping*, bloqueio no acesso a periféricos, escalonamento que favorece a distribuição equitativa do CPU, etc.)
- *Soft Real-Time* - *OS9* que é dito *Unix like*, proprietário (*RadiSys Corporation*) que usam técnicas de tempo real (exclusão de memória virtual, mecanismos de IPC rápidos e com bloqueios reduzidos, chamadas ao sistema curtas) mas não oferecem garantias temporais (tipo *best-e"ort*)
- *Hard Real-Time* - *SHaRK*, *RTLinux*, *QNX*, *RTAI* - Oferecem garantias temporais e muitos oferecem modificações no *kernel* original do *Linux* para modificar e incluir *microkernel* "superior" (escalador superior que escalona o escalonador tradicional do *Linux*).

Escalonador de Tarefas de Tempo Real

O escalonador é um serviço do gerenciador de processos. Ele é responsável por decidir quando uma tarefa (processo, *thread*) usa os recursos do sistema (tempo de CPU, canais de comunicação, etc.).

Normalmente esta decisão leva em conta o *load balance* (balanceamento de carga) para que o sistema atinja efetivamente uma qualidade de serviço. A necessidade de um algorítimo de escalonamento (*scheduling algorithm*) se dá pela natureza dos sistemas modernos que oferecem serviços de multitarefas e multiplexação.



O escalonador (*scheduler*) se "preocupa" com:

- *Throughput* – o número total de processos que completam sua execução por unidade de tempo. Latência (*latency*), especificamente:
  - Tempo de retorno (*turnaround time*) – tempo total entre submissão de um processo e sua terminação. Tempo de resposta (*response time*) – quantidade de tempo que leva desde uma solicitação ser submetida até que a primeira resposta seja produzida.
- Tempo de espera (*waiting time*) – igual ao tempo de CPU para cada prioridade de processo. Também é o tempo em que um processo fica esperando na fila para ser processado.

Na prática os objetivos do escalonador frequentemente são conflitantes (ex. máximo *throughput* com mínima latência). Assim um escalonador implementa um algoritmo que garante um rendimento mínimo com métricas. Em sistemas tempo real, o escalonador deve também garantir que processos cumpram seus prazos (*deadlines*). Isto é crucial para manter os sistemas (de controle) estáveis.

Regras de Escalonamento

São algoritmos usados para distribuir recursos entre partes que simultaneamente e assincronamente os requerem. Possuem várias aplicações:

- Roteadores – gerenciamento de tráfego de pacotes
- SO – gerenciamento de tempo de CPU entre processos e *threads* Drives
- de disco (*I/O scheduling*)
- Impressoras (*print spooler*) etc
-

O principal propósito é minimizar o tempo de espera por recursos (*starvation*) e garantir a competição limpa (*fairness*) entre as partes (processos na fila).

Algorimos de Escalonamento

FIFO – First In First Out

Também conhecido como *First Come, First Served* (FCFS), é o mais simples algoritmo de escalonamento. Simplesmente coloca os processos fila na ordem que eles "chegam". Os chaveamentos de contexto somente ocorrem no final do processo. O *overhead* é mínimo, pois não há reorganização (da fila). O *throughput* pode ser baixo, pois processos "grandes" podem usar por bastante tempo a CPU. O mesmo pode ocorrer com o *turnaround time*, *waiting time* e *response time* pelos mesmos motivos.

Não há tratamento de prioridades, logo este algoritmo tem problemas para cumprir *deadlines*. A falta de tratamento de prioridades significa que se há garantias que os processos irão eventualmente terminar, não haverá *starvation*. Se não houver esta garantia, se um processo não terminar, haverá *starvation*. Resumindo, é baseado em "fila" (*queuing*).

Shortest Job First (SJF)

Com esta estratégia o escalonador organiza o processo com o menor tempo estimado de processamento será colocado na primeira posição da fila e assim por diante. Isto requer um prévio conhecimento ou estimação do tempo de término de um processo. Se um processo "menor" chegar durante a execução de um processo maior, este último pode ser interrompido (preempção). Isto gera um *overhead* através de chaveamentos de contexto excessivos.

O escalonador também deve colocar cada processo em um lugar específico na fila, criando um *overhead* adicional. Este algoritmo é projetado para um máximo de *throughput* na maioria dos casos. Tempos de espera e de resposta aumentam conforme os requisitos computacionais do processo aumentam. Já que o *turnaround time* é baseado no tempo de espera mais o tempo de processamento, processos longos são significativamente afetados.

O tempo de espera geral é menor que no FIFO, já que processos menores não tem que esperar a conclusão de processos maiores. *Starvation* é possível, especialmente em sistemas com muitos processos pequenos.

Uma algoritmo parecido é o *Shortest Remaining Time* (SRT), onde a cada intervalo (unidade de tempo) é verificado qual o processo tem menos tempo restante para finalizar seu processamento. Este processo recebe então a primeira posição na fila.

Fixed priority pre-emptive scheduling

SOs assinalam níveis de prioridade (fixos) para cada processo. O escalonador os organiza na fila de acordo com a prioridade. Processos de prioridade menor são interrompidos (preempção) por processos de maior prioridade. O *overhead* não é mínimo e nem insignificante. Tem um vantagem particular em termos de *throughput* sobre o FIFO. O tempo de espera e de resposta dependem da prioridade do processo.

Deadlines podem ser cumpridos assinalando altas prioridades a processos com *deadline*. *Starvation* de processos de baixa prioridade são possíveis se houver um grande número de processos de alta prioridade na fila.

Round-Robin Scheduling

Divide o tempo de processamento em unidades de tempo (*time quantum*) iguais para cada processo na fila. Envolve grande *overhead*, especialmente com unidades de tempo pequenas. Possui um balanceamento de *throughput* entre o FCFS e SJF, pois processos menores são completados mais rapidamente que no FCFS e processos maiores são completados antes do SJF.

Possui um baixo tempo médio de resposta, pois o tempo de espera é dependente do número de processos e não do tamanho médio do processo.

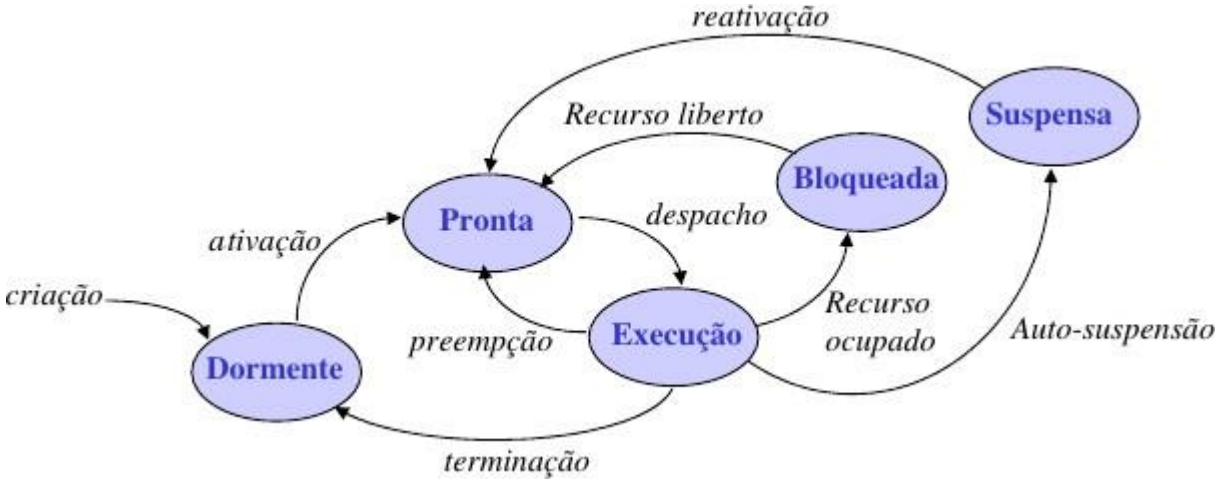
Por causa do tempo de espera alto os *deadlines* são raramente respeitados. *Starvation* pode nunca ocorrer, já que não há prioridades. A ordem de alocação de unidades de tempo é baseado no tempo de chegada do processo similarmente ao FCFS.

Multilevel queue scheduling

É usado em situações em que os processos são facilmente divididos em diferentes grupos. Por exemplo uma divisão simples é entre processos que requerem interatividade (em primeiro plano) e processos em lote (em segundo plano). Este diferentes processos tem requisitos de tempos de resposta diferentes e portanto tem diferentes necessidades de escalonamento. Este algoritmo é muito útil para problemas de memória compartilhada.

A maioria do SOs tradicionais usa uma variante de *multilevel queue scheduling* chamada de *multilevel feedback queue*.

Para se ter uma noção do trabalho do escalonador é importante saber os estados de uma tarefa. A figura abaixo mostra os estados e as ações que provocam a mudança de estado das tarefas.



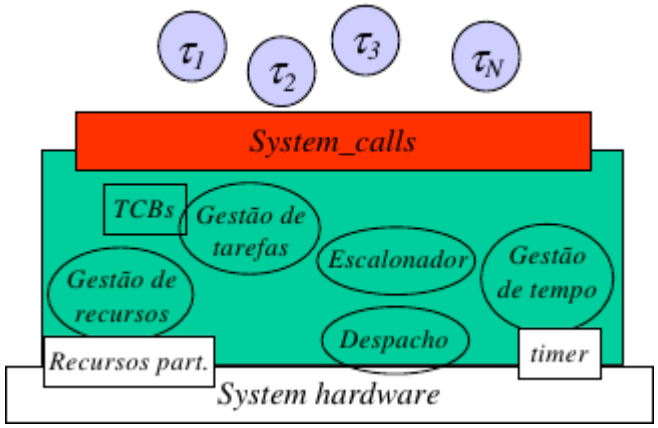
A transição do estado pronta para execução e o inverso esta ligada diretamente ao escalonador. Existem ações que a própria tarefa dispara, como: auto-suspensão e terminação. As outras ações são serviços do gerenciador de tarefas. A organização da ativação (estado dormente para pronta) também é serviço do escalonador já que é este que organiza a fila.

Arquitetura de um Kernel RT

Um *kernel* tempo real deve prestar os seguintes serviços básicos: Gerenciamento de

- tarefas (criação , destruição, ativação inicial, estado)
- Gerenciamento do tempo (ativações, policiamento, medição de intervalos)
- Escalonamento de tarefas (escolha da tarefa a executar)
- Despacho de tarefas (colocação em execução)
- Gerenciamento de recursos partilhados (mutexes, semáforos, monitores)

Para suprir estes serviços o *kernel* está estruturado como mostra a figura abaixo.



Escalonamento para Tempo Real

Em tempo real deve-se usar um critério determinístico para permitir calcular o atraso máximo (pior caso) que uma tarefa pode sofrer na fila. Um escalonamento diz-se praticável (*feasible schedule*) se cumpre as restrições associadas ao conjunto de tarefas (temporais, não preempção, recursos partilhados, precedências). Um conjunto de tarefas diz- se escalonável (*schedulable task set*) se existe pelo menos um escalonamento praticável para esse conjunto.

Um algoritmo de escalonamento pode ser:

- Preemptivo versus não-preemptivo Estático
- versus dinâmico
- O"-line versus on-line Ótimo
- versus heurístico
- Com garantias de pior caso versus *best e"ort* (melhor possível)

A seguir veremos algoritmos de escalonamento exclusivos para tempo real.

Rate Monotonic

Quanto menor o período do processo, maior a prioridade. Diz-se que o escalonamento é ótimo para D (*deadline*) = T (período).

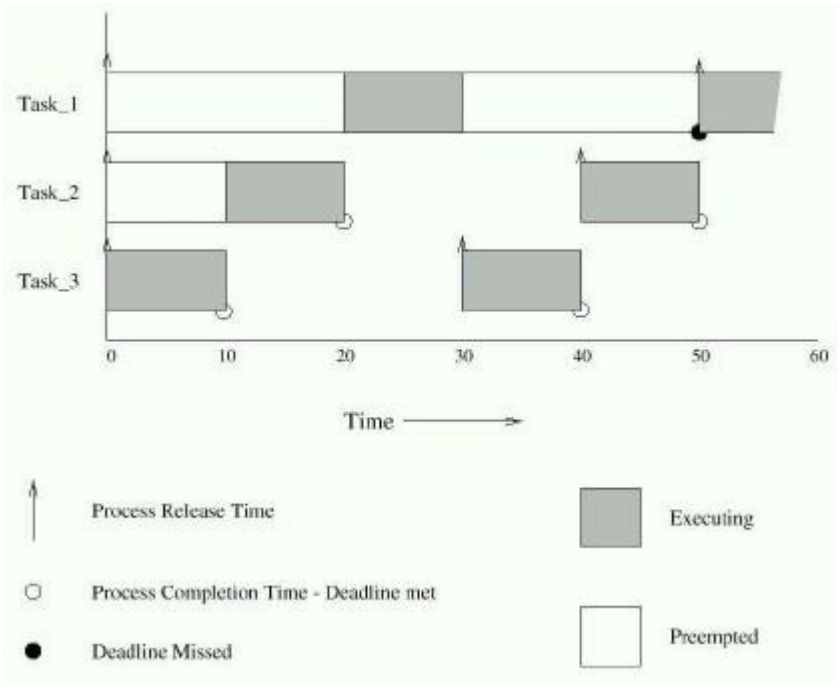
Se um conjunto de processos pode ser escalonado com um esquema de prioridades fixas, este conjunto de processos também pode ser escalonado com *rate monotonic*.

Exemplo de *Rate Monotonic*:

Para um conjunto de 3 tarefas com as seguintes características.

	Período (T)	Tempo de Computação (C)	Prioridade (P)
Task_1	50 ms	12 ms	*1*
Task_2	40 ms	10 ms	*2*
Task_3	30 ms	10 ms	*3*

Para a execução teríamos a seguinte sequência ilustrada na figura abaixo.



Note que no instante t = 50 ms, a Task_1 executou apenas 10 ms, quando necessitava executar 12 ms, ou seja, houve perda de processamento, já que no período de 50 ms não pode ser executado os 12 ms de tempo de processamento requerido pela tarefa.

Deadline Monotonic

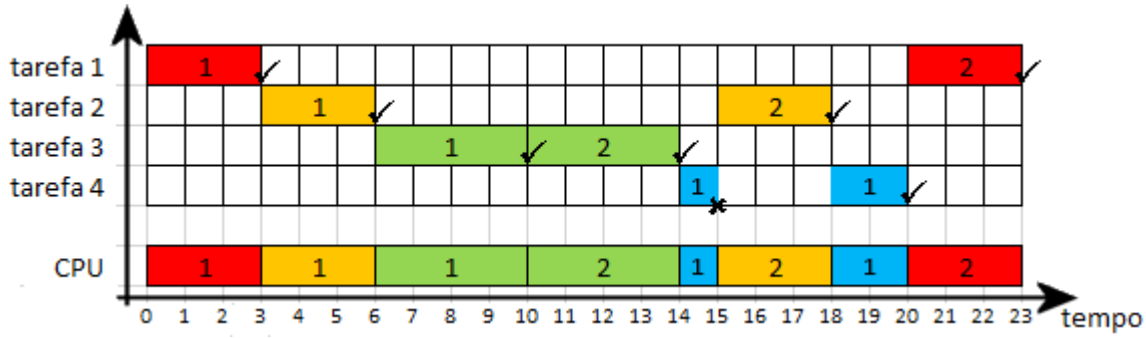
Neste algoritmo quanto menor o *deadline* do processo, maior a prioridade. Escalonamento é ótimo para D (*deadline*) < T (período).

Exemplo de *Deadline Monotonic*:

Para um conjunto de 4 tarefas com as seguintes características.

	Período (T)	Deadline (ms)	Tempo de Computação (C)	Prioridade (P)
Task_1	20 ms	5 ms	3 ms	*4*
Task_2	15 ms	7 ms	3 ms	*3*
Task_3	10 ms	10 ms	4 ms	*2*
Task_4	20 ms	20 ms	3 ms	*1*

A figura abaixo ilustra o escalonamento para este conjunto de tarefas, com a unidade de tempo mínima de 1 ms.



Percebe-se que a tarefa 4 é interrompida no instante 15 ms, isto é, há a preempção pelo fato de neste instante entrar na fila a segunda execução (2° ciclo) da tarefa 2 que possui maior prioridade. Quando a tarefa 2 termina em 18 ms, a tarefa 4 é retomada e termina sua execução exatamente no instante máximo de seu *deadline*, em 20 ms.

Se fizermos o escalonamento por 2 ciclos da tarefa de maior período, isto é, 40 ms, se notará que o conjunto obedece todos os *deadlines* e portanto podemos dizer que o conjunto é praticável (*feasible scheduling*) e o conjunto de tarefas é escalonável (*schedulable task set*).

Earliest Deadline First

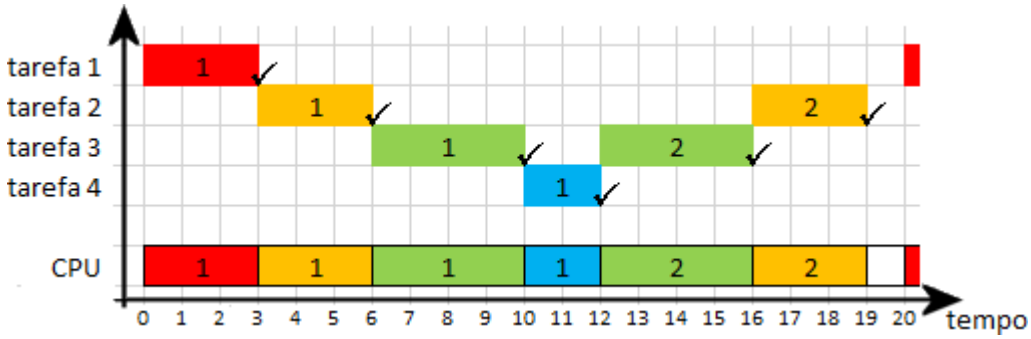
Usado em tarefas de única instância ou periódicas, assíncronas, preemptivas. Executar em cada instante a tarefa com *deadline* mais próximo.

Na prática este é o algoritmo mais utilizado.

Para um conjunto de 4 tarefas com as seguintes características.

	Período (T)	Deadline (ms)	Tempo de Computação (C)
Task_1	20 ms	5 ms 5 ms 7 ms	3 ms
Task_2	15 ms	7 ms	3 ms
Task_3	10 ms	10 ms 10 ms	4 ms
Task_4	20 ms	18 ms 18 ms	2 ms

A figura abaixo ilustra o escalonamento para este conjunto de tarefas, com a unidade de tempo mínima de 1 ms.



Percebe-se que diferente do Deadline Monotonic a tarefa 4 não é "perde" no instante 10 ms, pois há uma tarefa, no caso a tarefa 4 que tem o deadline mais próximo 18 ms (restando 8 ms) se comparado com o 2° ciclo da tarefa 3 que neste mesmo instante (10 ms) tem deadline até 20 ms (restanto 10 ms).

Se fizermos o escalonamento por 2 ciclos da tarefa de maior período, isto é, 36 ms, se notará que o conjunto obedece todos os *deadlines* e portanto podemos dizer que o conjunto é praticável (*feasible scheduling*) e o conjunto de tarefas é escalonável (*schedulable task set*).

Sistemas Operacionais e Tempo Real 2020-2

Mecanismos de Sincronização e Comunicação entre Tarefas

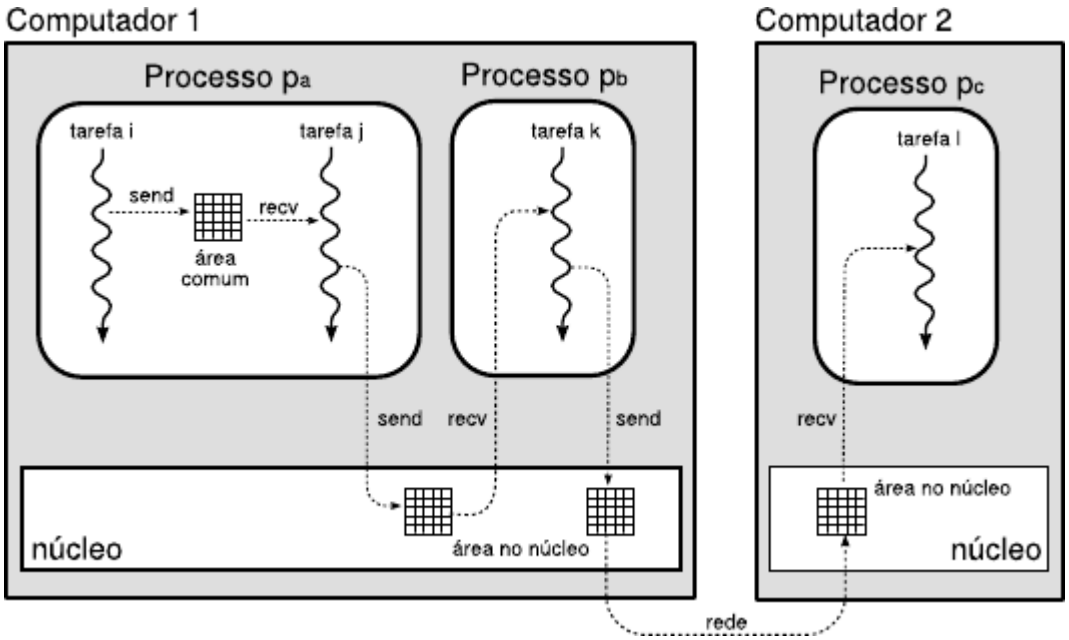
Implementaracomunicaçãoentretarefaspodesersimplesoucomplexo,dependendodasituação. Seastarefas estãonomesmoprocesso,elas compartilhamamesmaáreadememóriaeacomunicaçãopodeentãoser
implementada facilmente, usando variáveis globais comuns.

Entretanto, caso as tarefas pertençam a processos distintos, não existem variáveis compartilhadas; neste caso, a comunicação tem de ser feita por intermédio do *kernel* do SO através do serviço de IPC (*Inter-Process Communication*), usando chamadas de sistema.

Existem 3 variantes da comunicação entre processos: intraprocesso (t_i

- $\rightarrow t_j$)
- interprocessos ($t_j \rightarrow t_k$) intersistemas($t_k \rightarrow$
- t_l)

A figura abaixo ilustra estas 3 variações.

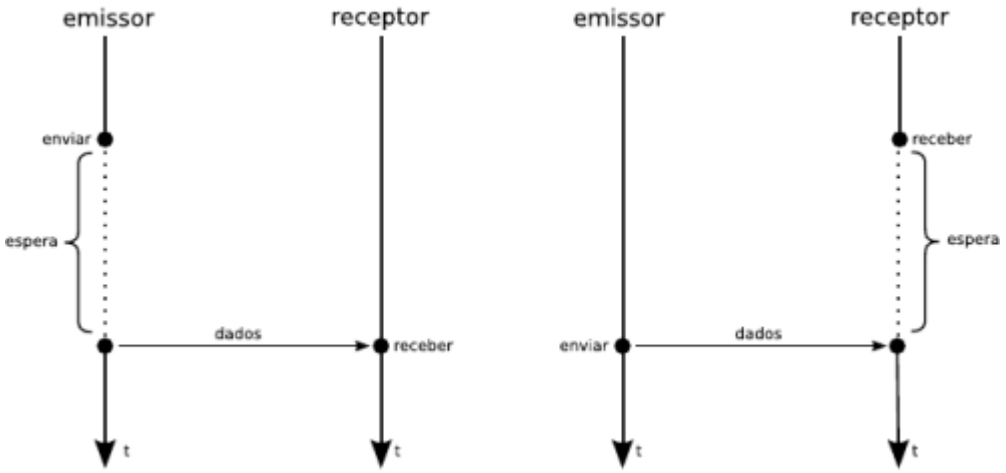


As comunicações entre processo podem ainda ser:

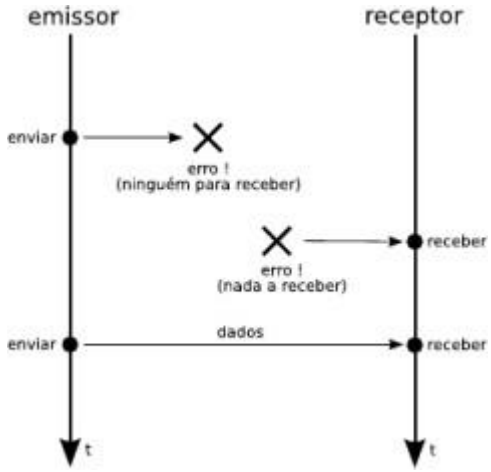
- Diretas - o emissor identifica claramente o receptor e vice-versa. Há 2 primitivas básicas: enviar (dados, destino), receber (dados,origem)
- Indiretas-maisflexíveis.Oemissorereceptornãoprecisamseconhecer,poisnãointeragemdiretamente. Nãohá informaçãodequemenviouos dados.Orelacionamentoéatravésdeumcanaldecomunicação.Há2 primitivas básicas: enviar (dados, canal), receber (dados, canal).

Emrelaçãoaosaspectosdesincronismodocanaldecomunicação,acomunicaçãoentretarefaspodeser:

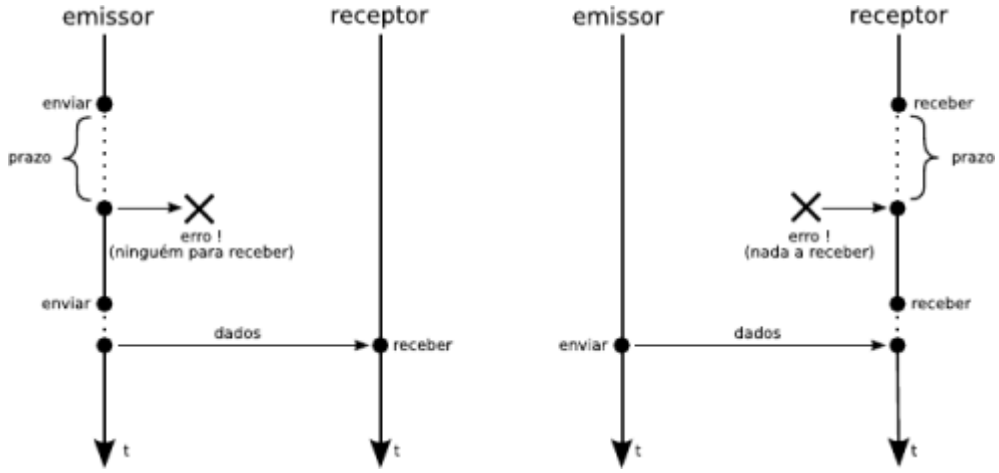
- Síncrona-quandoasoperaçõesdeenvioerecepçãodedadosbloqueiam(suspendem)astarefasenvolvidas atéaconclusãodacomunicação:o emissorserábloqueadoatéqueainformaçãosejarecebidapeloreceptor, e vice-versa.



- Assíncrona - as primitivas de envio e recepção não são bloqueantes: caso a comunicação não seja possível no momento em que cada operação é invocada, esta retorna imediatamente com uma indicação de erro. Se transmissor e receptor possuem "períodos" diferentes (ambos operem assincronamente) há a necessidade de uso de um *buffer*.



- Semi-síncrona - primitivas de comunicação semi-síncronas têm um comportamento síncrono (bloqueante) durante um prazo pré-definido. Caso o prazo se esgote sem que a comunicação tenha ocorrido, a primitiva retorna com uma indicação de erro. Para refletir esse comportamento, as primitivas de comunicação recebem um parâmetro adicional: enviar(dados, destino, prazo) e receber(dados, origem, prazo).



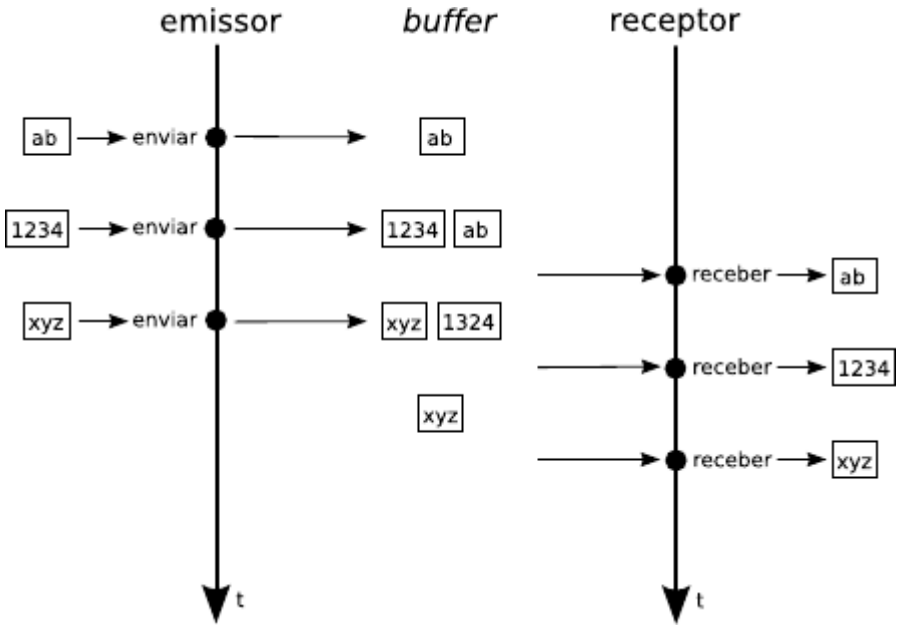
Com relação ao formato de envio na comunicação, a informação enviada pelo emissor ao receptor pode ser vista basicamente de duas formas, como:

- uma sequência de mensagens independentes, cada uma com seu próprio conteúdo, ou
- um fluxo sequencial e contínuo de dados, imitando o comportamento de um arquivo com acesso sequencial.

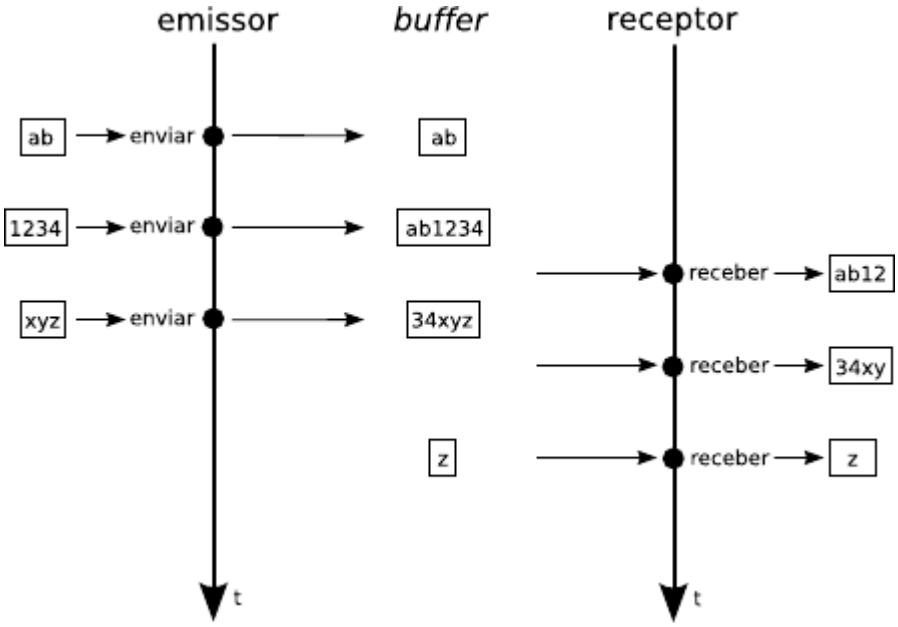
Na abordagem baseada em mensagens, cada mensagem consiste de um pacote de dados que pode ser "tipado" ou não. Esse pacote é recebido ou descartado pelo receptor em sua íntegra; não existe a possibilidade de receber "meia mensagem". Servem de implementações exemplo para esta abordagem as *message queues* do *Unix* e os protocolos de rede IP e UDP.

Caso a comunicação seja definida como um fluxo contínuo de dados, o canal de comunicação é visto como o equivalente a um arquivo, onde respeita-se a ordem de envio dos dados. Não há separação lógica entre os dados enviados em operações separadas, ou seja, eles podem ser lidos byte a byte ou em grandes blocos a cada operação de recepção, a critério do receptor. Servem de implementações exemplo para esta abordagem os *pipes* do *Unix* e protocolo TCP/IP.

A figura baixo ilustra formato baseado em mensagens.

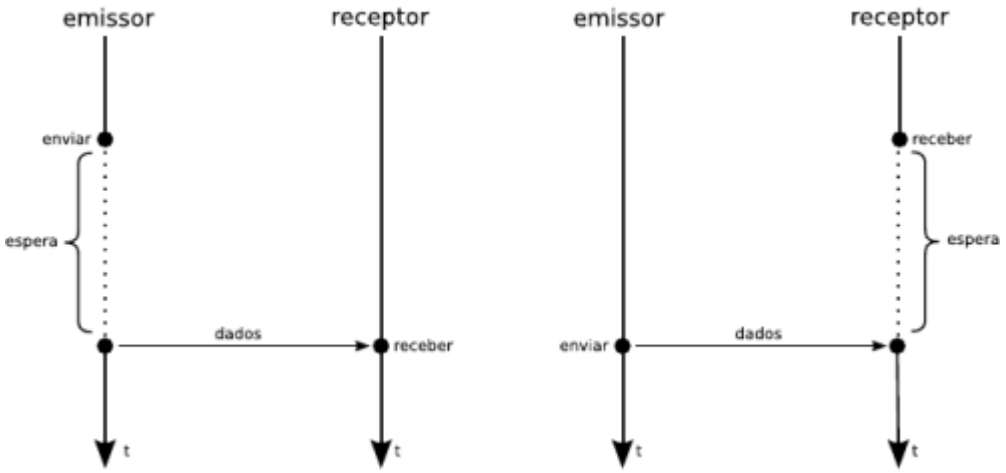


Em contradição ao formato baseado em mensagens a figura baixo ilustra o formato baseado em fluxo de dados.

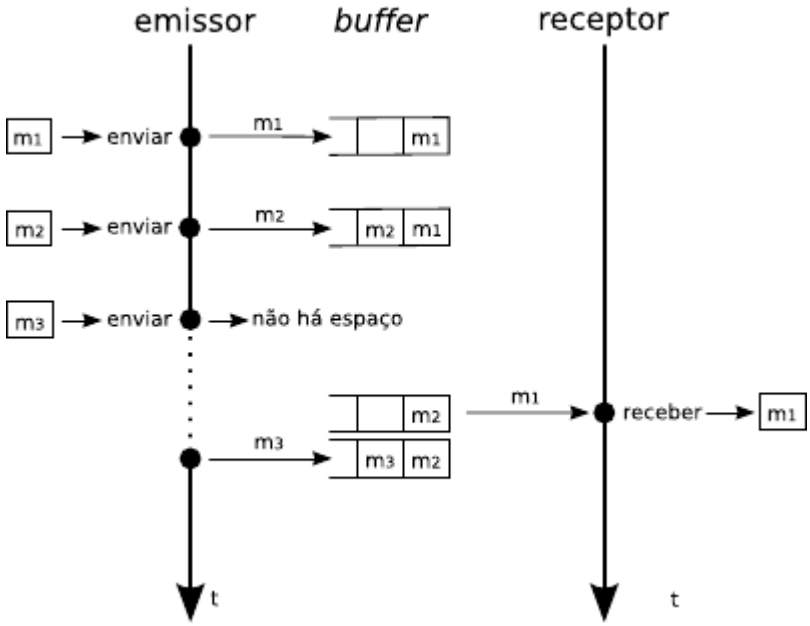


O comportamento síncrono ou assíncrono de um canal de comunicação pode ser afetado pela presença de *bu"ers* que permitam armazenar temporariamente os dados em trânsito. Em relação à capacidade de *bu"ering* do canal de comunicação, três situações devem ser analisadas:

- capacidade nula - canal não armazena dados, não pode ser usado comunicação assíncrona. É também chamado de *Rendez-Vous*. Só pode ser empregado em comunicação síncrona onde haverá bloqueio.

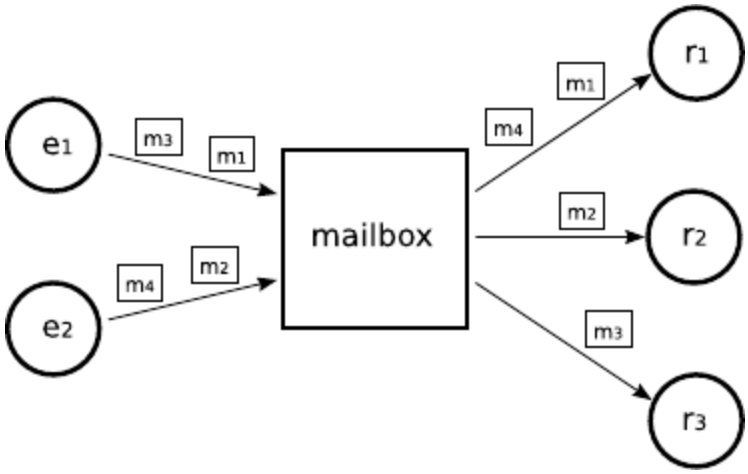


- capacidade infinita - o emissor sempre pode enviar dados, que serão armazenados no *buffer* do canal enquanto o receptor não os consumir. É uma utopia, já que não se pode ter um *buffer* de capacidade infinita. capacidade finita - uma quantidade finita (n) de dados pode ser enviada pelo emissor sem que o receptor os consuma. O exemplo abaixo mostra um canal de capacidade 2.

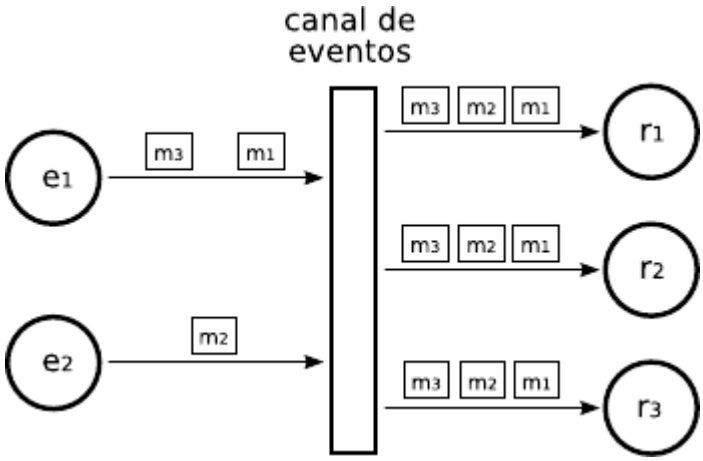


Existem situações em que uma tarefa necessita comunicar com várias outras, como por exemplo em sistemas de *chat* ou mensagens instantâneas. Dessa forma, os mecanismos de comunicação também podem ser classificados de acordo com o número de tarefas participantes:

- 1:1 - quando exatamente um emissor e um receptor interagem através do canal de comunicação M:N: quando um ou mais emissores enviam mensagens para um ou mais receptores. Duas situações distintas:
  - Cada mensagem é recebida por apenas um receptor (em geral aquele que pedir primeiro) - chamado de abordagem *mailbox*. É implementado nas *message queues* do *Unix* e *sockets* UDP.



- Cadamensagemérecebidaportodososreceptores(cadareceptorrecebeumacópiadamensagem)- chamadodeabordagemdifusão (multicast)oudecanaldeeventos.Éimplementadonoprotocolo UDP.



Comovimosnasaulaspráticasaexecuçãode*threads* concorrentesé"aleatória"(nãosesabequandoenema ordemqueserãoexecutadas).Seas*threads* tiveremquecompartilharousecomunicar,háanecessidadedeseusar mecanismos de sincronização. Estes mecanismos servem para organizar o acesso a dados e recursos compartilhados, como:

- Memória compartilhada Arquivos
- Dispositivos de I/O Etc.

No *Unix* existem dois tipos de mecanismos de sincronização: os mutexes e os semáforos.

Mutexescomoopróprionomesugerepermite"exclusãomútua",ouseja,permiteacessoporsomenteumprocesso a uma área crítica (geralmente compartilhada).

OssemáforossãoumconceitointroduzidoporDijkstra(1968)noSOTHE.Sãomecanismosquesimplificamos protocolosparasincronização.Umsemáforoéumavariávelinteira,nãonegativa,naqualsepodefazerapenasduas operações (além dainicialização):

- wait(s) wait(s) -se s > 0 decrementa s, senão suspende o processo
- signal(s) signal(s) - se existe algum processo suspenso em s, libera um deles, senão incrementa s. Não é especificado qual processo é liberado e as operações nos semáforos são atômicas.

Como vimos durante a disciplina há vários SOTRs disponíveis:

- OS9
- QNX
- Sha
- RK
- RTL
- inu
  x
  Xen
  om
  ai
  RTA
  I

Muitos oferecem modificações no *kernel* original do Linux para modificar e incluir microkernel “superior”.

Uma lista completa pode ser vista em https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems No

RTLinux, no *Xenomai* e no *RTAI* um *kernel* tempo real coexiste com o *kernel* do Linux. O objetivo deste modelo é possibilitar que aplicações utilizem serviços do *Linux* como comunicação por rede, sistemas de arquivos, controle de processos, etc. O Linux é responsável até mesmo pela inicialização do *kernel* tempo real e pelos *drivers* deste. A instalação destas modificações no *kernel* se dá através da aplicação de um *patch.* Outras funcionalidades são através de módulos carregáveis.

Exemplos:

- Xenomai na Raspberry Pi
- Xenomai na BeagleBone - https://www.youtube.com/watch?v=Cnfx5QdNJog
- RTAI no Ubuntu - https://www.youtube.com/watch?v=Gnyf_xf6kig
- ARTe (Arduino Real-Time extension
- ChibiOS para Arduino

Em sistemas distribuídos o que interconecta esta distribuição é a infraestrutura de rede.

Assim, para serem usadas técnicas de tempo real deve-se ter uma comunicação determinística entre os sistemas; Ainda

devem-se considerar situações onde a comunicação é perdida.

Vimos algumas linguagens com suporte a tempo real, como é o caso do ADA, ilustrado na figura.

Using a Timing_Event as a "watchdog timer"

```
1. ...
2. protected Alarm is
3.    pragma Priority (System.Interrupt_Priority'Last);
4.    procedure Respond (E : in out Timing_Event);
5. end Alarm;
6.
7. ...
8.
9. Watchdog : Timing_Event;
10.
11. task Worker is
12.    pragma Priority (...);
13. end Worker;
14.
15. task body Worker is
16.    Cancelled : Boolean;
17. begin
18.    loop
19.       ...
20.       Watchdog.Set_Handler
21.              (In_Time => Milliseconds (50),
22.              Handler => Alarm.Respond'Access);
23.       Do_Critical_Work;
24.       Watchdog.Cancel_Handler (Cancelled);
25.       ...
26.    end loop;
27. end Worker;
```

Há outras linguagens que oferecem suporte a tempo real como o Java e o C/C++. Um exemplo de código C para uso no RTAI pode ser visto abaixo.

Sistemas Operacionais e Tempo Real 2020-2

Comunicação em Sistemas Distribuídos

Já vimos o Modelo Cliente-Servidor em Redes Industriais o que remete diretamente em *sockets*, RMI e RPC. Assim, como

já vimos *sockets* iremos focar em RMI e RPC.

Threads também já foram vistas anteriormente.

RPC

O RPC (*Remote Procedure Call*) é uma forma de comunicação entre processos que permite que um programa computacional execute uma subrotina ou procedimento em outro address space (geralmente outro computador numa rede compartilhada) sem a necessidade de codificar os detalhes desta interação remota;

Ou seja, o programador escreve o mesmo código seja a subrotina local ou remota. O RPC é uma tecnologia popular para a implementação do modelo cliente-servidor de computação distribuída.

Uma chamada de procedimento remoto é iniciada pelo cliente enviando uma mensagem para um servidor remoto para executar um procedimento específico. Uma resposta é retornada ao cliente.
Uma diferença importante entre chamadas de procedimento remotas e chamadas de procedimento locais é que, no primeiro caso, a chamada pode falhar por problemas da rede. Nesse caso, não há nem mesmo garantia de que o procedimento foi invocado.

A ideia de RPC data de 1976, quando foi descrito no RFC 707 (*A High-Level Framework for Network-Based Resource Sharing*).

Um dos primeiros usos comerciais da tecnologia foi feita pela Xerox no "Courier", de 1981. A primeira implementação popular para *Unix* foi o Sun RPC (atualmente chamado ONC RPC), usado como base do *Network File System* e que ainda é usada em diversas plataformas.

Outra implementação pioneira em *Unix* foi o *Network Computing System* (NCS) da Apollo Computer, que posteriormente foi usada como fundação do DCE/RPC no *Distributed Computing Environment* (DCE).

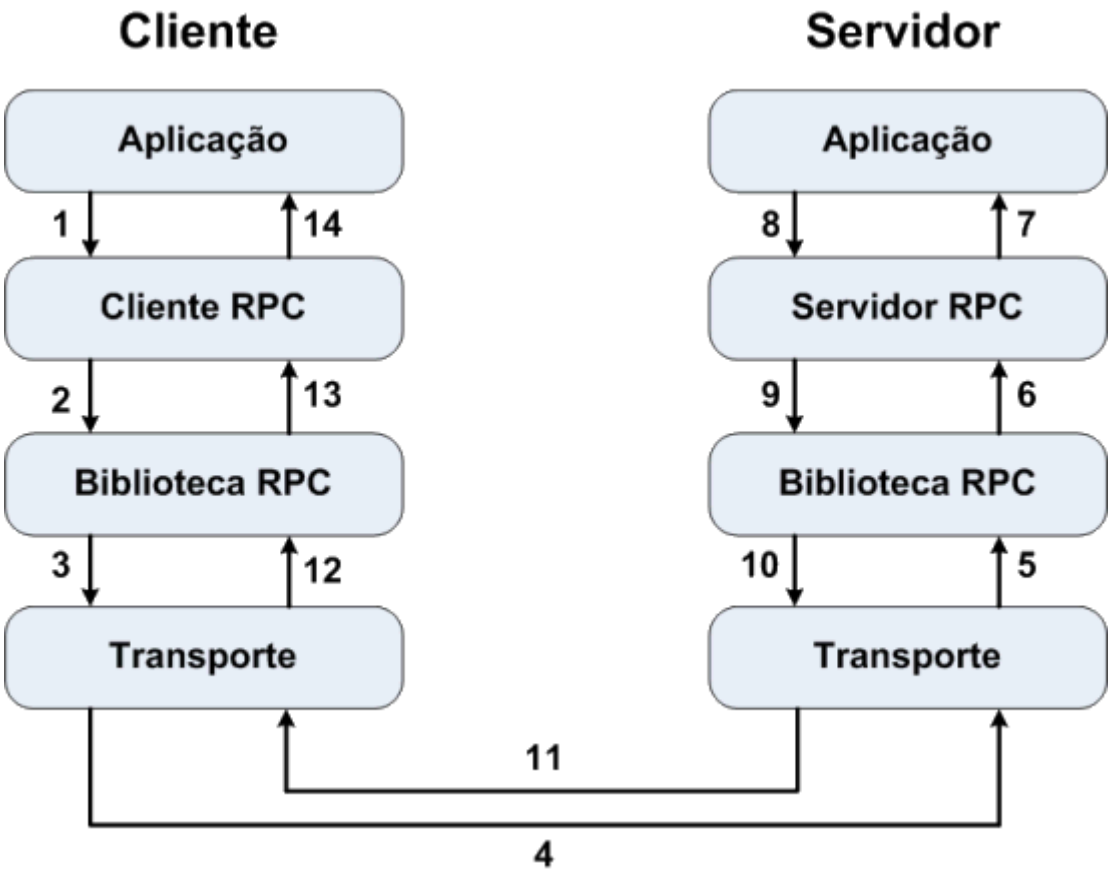
Uma década depois a Microsoft adotou o DCE/RPC como base para a sua própria implementação de RPC, MSRPC, a DCOM foi implementada com base nesse sistema.

Atualmente utiliza-se XML como linguagem de descrição de interface e HTTP como protocolo de rede para formar serviços Web, cujas implementações incluem SOAP e XML-RPC.

O modelo de RPC é similar ao modelo de chamadas locais de procedimentos, no qual a rotina que invoca o procedimento coloca os argumentos em uma área de memória bem conhecida e transfere o controle para o procedimento em execução, que lê os argumentos e os processa. Em algum momento, a rotina retoma o controle, extraindo o resultado da execução de uma área bem conhecida da memória. Após isso, a rotina prossegue com a execução normal.

No modelo RPC, uma thread é responsável pelo controle de dois processos: invocador e servidor. O processo invocador primeiro manda uma mensagem para o processo servidor e aguarda (bloqueia) uma mensagem de resposta. A mensagem de invocação contém os parâmetros do procedimento e a mensagem de resposta contém o resultado da execução do procedimento. Uma vez que a mensagem de resposta é recebida, os resultados da execução do procedimento são coletados e a execução do invocador prossegue.

Do lado do servidor, um processo permanece em espera até a chegada de uma mensagem de invocação. Quando uma mensagem de invocação é recebida, o servidor extrai os parâmetros, processa-os e produz os resultados, que são enviados na mensagem de resposta. O servidor, então, volta a esperar por uma nova mensagem de invocação.



Abaixo está o exemplo de um servidor e um cliente XML-RPC.

```
// servidor XML-RPC

import org.apache.xmlrpc.*;

public class JavaServer {

 public Integer sum(int x, int y) {
    return new Integer(x+y);
 }

 public static void main (String [] args) {
  try {

     System.out.println("Attempting to start XML-RPC Server...");
     WebServer server = new WebServer(80);
     server.addHandler("sample", new JavaServer());
     server.start();
     System.out.println("Started successfully.");
     System.out.println("Accepting requests. (Halt program to stop.)");
   } catch (Exception exception) {
     System.err.println("JavaServer: " + exception);
   }
  }
 }
```

```
// cliente XML-RPC

import java.util.*;
import org.apache.xmlrpc.*;

public class JavaClient {
 public static void main (String [] args) {
  try {

     XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");
     Vector params = new Vector();
     params.addElement(new Integer(17));
     params.addElement(new Integer(13));

     Object result = server.execute("sample.sum", params);

     int sum = ((Integer) result).intValue();
     System.out.println("The sum is: "+ sum);

   } catch (Exception exception) {
     System.err.println("JavaClient: " + exception);
   }
  }
}
```

Nota-se no cliente a chamada a "função" (procedimento remoto) `sum` do servidor, que é executado como se fosse local.

RMI

O RMI (*Remote Method Invocation*) é uma interface de programação que permite a execução de chamadas remotas no estilo RPC em aplicações desenvolvidas em Java. É uma das abordagens da plataforma Java para prover as funcionalidades de uma plataforma de objetos distribuídos.

Esse sistema de objetos distribuídos faz parte do núcleo básico de Java desde a versão JDK 1.1, com sua API sendo especificada através do pacote java.rmi e seus subpacotes. Através da utilização da arquitetura RMI, é possível que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais.

O funcionamento de RMI consiste basicamente em dois programas, segundo a arquitetura cliente-servidor. O servidor instancia objetos remotos, o referencia com um nome e faz um ***bind*** dele numa porta, onde este objeto espera por clientes que invoquem seus métodos.

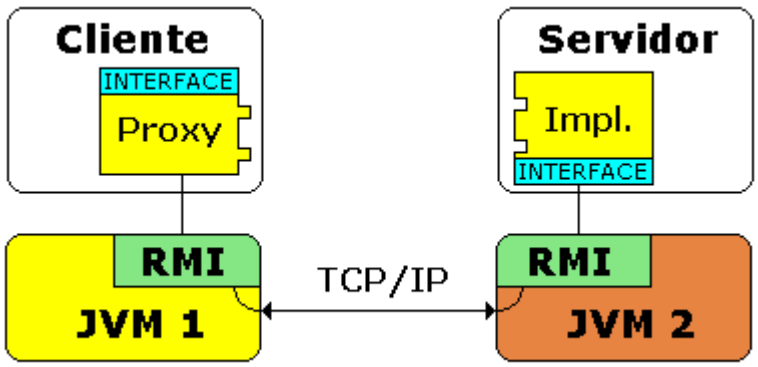
Já o cliente referencia remotamente um ou mais métodos de um objeto remoto. O RMI fornece os mecanismos para que a comunicação entre cliente e servidor seja possível. Esse tipo de aplicação geralmente é denominada como Aplicação de Objeto Distribuído.

Aplicações distribuídas precisam, portanto, executar as seguintes ações:

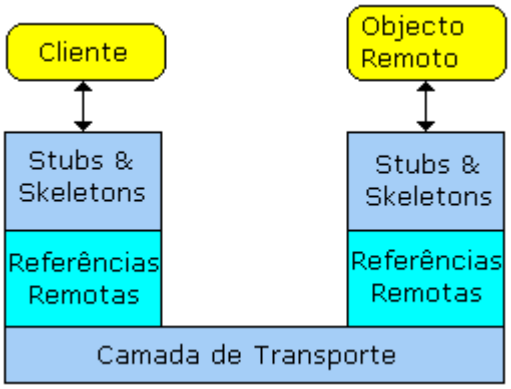
- Localizar Objetos Remotos - Uma aplicação pode usar dois mecanismos para obter referências de objetos remotos. Ela pode registrar o objeto remoto com a ferramenta de nomes do RMI, que se chama "rmiregistry", ou ela pode passar e retornar referências aos objetos remotos como parte de sua operação.
- Comunicar com Objetos Remotos - Os detalhes de comunicação entre objetos remotos são tratados pelo RMI, ou seja, para o programador, a comunicação remota é semelhante a uma chamada ao método localmente.
- Carregar "bytecodes" de objetos móveis - Como o RMI permite que objetos remotos sejam passados como parâmetros numa função, ele fornece os mecanismos necessários para carregar o código dos objetos remotos.

Uma das principais vantagens do RMI é sua capacidade de baixar o código de um objeto, caso a classe desse objeto não seja definida pela máquina virtual do receptor. Os tipos e o comportamento de um objeto, previamente disponíveis apenas em uma máquina virtual, agora podem ser transmitidos para outra máquina virtual, possivelmente remota.

Essa funcionalidade do RMI permite que o código da aplicação seja atualizado dinamicamente, sem a necessidade de recompilar o código. Outra vantagem do RMI é sua segurança intrínseca, pois apenas *hosts* registrados podem invocar métodos remotos. A figura abaixo ilustra o model RMI.



No RMI há ainda o conceito de *stubs* e *skeletons* como mostra a figura.



A camada mais próxima do programador, ou seja da aplicação cliente e do objecto remoto, é a camada *Stubs/Skeletons*.

Os *Stubs* são classes usadas do lado da aplicação cliente e funcionam como *proxies* (intermediadores) entre a aplicação cliente e o objecto remoto. Eles recebem os parâmetros dos métodos exportados pelo objeto remoto (definidas pela interface da classe remota) e reencaminham-nos para o lado do servidor onde serão interpretados por uma instância de uma classe *Skeleton*.

O *Skeleton* recebe os parâmetros enviados pelo *Stub* e executa as respectivas chamadas no objecto remoto. Eles também são responsáveis por receber o valor de retorno do método remoto (local na sua perspectiva) e direcioná- los para os *Stubs* dos clientes correspondentes.

No RMI ainda há o uso de uma classe para descrever a interface (entre servidor e cliente RMI). Também existe um compilador específico para criar *stubs* e *skeletons*, o RMIC do pacote JSDK. Para uso do RMI antes há a necessidade de registrar o servico com o RMIREGISTRY.

Middleware de Computação Distribuída

Middleware como o próprio nome sugere é um *software* que faz o “meio de campo”. Em computação distribuída nem sempre a arquitetura dos “dispositivos distribuídos” é a mesma gerando uma incompatibilidade.

Para solucionar isto existe implementações que harmonizam estas diferenças possibilitando a interoperabilidade. Implementações clássicas de middleware para este propósito são o CORBA, DCOM e o J2EE.

O CORBA, *Common Object Request Broker Architecture*, é um padrão definido pela *Object Management Group* (OMG) que possibilita que componentes de software escritos em linguagens diferentes e rodando em arquiteturas diferentes se tornem interoperáveis. O CORBA suporta a múltiplas plataformas e é um dos modelos mais populares de objetos distribuídos, juntamente com o DCOM, formato proprietário da Microsoft.

O CORBA utiliza a IDL (*Interface Definition Language*), uma linguagem baseada em C++ que não possui algoritmos nem variáveis, ou seja, é puramente declarativa, e, portanto, é independente da linguagem de programação utilizada para acessá-la.

Há o padrão de IDL definido pelo OMG para C, C++, Java, TTCN, COBOL, Smalltalk, Ada, Lisp, C#, Python e IDLscript. Possibilita a interoperabilidade entre os diversos sistemas, visto há separação entre interface e execução. A interface de cada objeto é definida de forma bastante específica, enquanto a sua execução (código fonte e dados) permanece oculta para o resto do sistema.

O DCOM, *Distributed Component Object Model*, é uma tecnologia proprietária da Microsoft para criação de componentes de software distribuídos em computadores interligados em rede. É uma extensão do COM (também da Microsoft) para a comunicação entre objetos em sistemas distribuídos. A tecnologia foi substituída, na plataforma de desenvolvimento .NET, pela API .NET Remoting e disponibilizada no WCF (*Windows Communication Foundation*).

O J2EE, *Java Platform Enterprise Edition*, é uma plataforma de programação para servidores na linguagem de programação Java. A plataforma inicialmente era conhecida por Java 2 Platform, Enterprise Edition ou J2EE, até ter seu nome trocado para Java EE na versão 5.0 que, posteriormente, foi chamada de Java EE 5. A versão atual é chamada de Java EE 6.

Difere-se da Plataforma Java Standard Edition (Java SE) pela adição de bibliotecas que fornecem funcionalidade para implementar software Java distribuído, tolerante a falhas e multicamada. É baseada amplamente em componentes modulares executando em um servidor de aplicações. A plataforma Java EE é considerada um padrão de desenvolvimento já que o fornecedor de software nesta plataforma deve seguir determinadas regras se quiser declarar os seus produtos como compatíveis com Java EE. Ela contém bibliotecas desenvolvidas para o acesso a base de dados, RPC, CORBA, etc. Devido a essas características a plataforma é utilizada principalmente para o desenvolvimento de aplicações corporativas.

O JEE contém bibliotecas desenvolvidas para o acesso a base de dados, RPC, CORBA, etc. Devido a essas características a plataforma é utilizada principalmente para o desenvolvimento de aplicações corporativas.

A plataforma JEE contém uma série de especificações e containers, cada uma com funcionalidades distintas: JDBC - *Java Database Connectivity*

- *Servlets* - são utilizados para o desenvolvimento de aplicações Web com conteúdo dinâmico. Ele contém uma API que abstrai e disponibiliza os recursos do servidor Web de maneira simplificada para o programador.
- JSP - *Java Server Pages* - uma especialização do servlet que permite que conteúdo dinâmico seja facilmente desenvolvido.
- JTA - *Java Transaction API* - é uma API que padroniza o tratamento de transações dentro de uma aplicação Java.
- EJBs - *Enterprise Java Beans* - utilizados no desenvolvimento de componentes de software. Eles permitem que o programador se concentre nas necessidades do negócio do cliente, enquanto questões de infraestrutura, segurança, disponibilidade e escalabilidade são responsabilidade do servidor de aplicações. JCA - *Java Connector Architecture* - é uma API que padroniza a ligação a aplicações legadas.
- JPA - *Java Persistence API* - é uma API que padroniza o acesso a banco de dados através de mapeamento Objeto/Relacional dos *Enterprise Java Beans*.

Sobre a programação de comunicação em sistemas distribuídos já tratamos sob programação básica, programação em sistemas de comunicação e agora vimos um pouco sob *middleware* o que facilita esta programação.