MC714

Sistemas Distribuídos

1° semestre, 2017

Primitivas para comunicação distribuída

Primitivas

- Primitivas de troca de mensagens: send() e receive()
- Envio de mensagem: send();
  - Pelo menos 2 parâmetros: destino e buffer com dados.
  - “Bufferizada” (buffer do kernel) ou não.
- Recepção de mensagem: receive();
  - Pelo menos 2 parâmetros: origem (pode ser *) e buffer de destino.

Primitivas

- Há duas maneiras de enviar dados quando a primitiva send é invocada: com buffer ou sem buffer.
  - Com buffer: copia os dados do espaço do usuário para um buffer do kernel. Depois, dados são copiados do buffer para a rede.
  - Sem buffer: dados são copiados diretamente do espaço do usuário para a rede.
- Para receive: geralmente com buffer porque dados podem já ter chegado quando a primitiva é invocada, precisando de um espaço temporário no kernel.

Síncrona / assíncrona

- Primitiva síncrona
  - Primitivas send() e receive() são síncronas se fazem *handshake.*
  - Processamento do *send* só termina depois que *receive* correspondente foi invocado e a operação de recebimento foi terminada.
  - Receive termina somente quando dados copiados ao buffer de recepção do usuário.
- Primitiva assíncrona
  - Primitiva *send* é assíncrona se controle retorna ao processo depois dos dados serem copiados para fora do buffer do usuário.
  - Não há sentido em definir uma primitiva receive assíncrona.

Bloqueante / não bloqueante

- Primitiva bloqueante
  - Controle retorna ao processo após o processamento da primitiva (síncrona ou assíncrona) ser completado.
- Primitiva não bloqueante
  - Controle retorna ao processo imediatamente após invocá-la, mesmo com a operação não terminada.
  - Para *send*, controle retorna ao processo antes da cópia dos dados para fora do buffer do usuário.
  - Para *receive*, controle pode retornar ao processo mesmo antes dos dados chegarem do remetente.

Bloqueante / não bloqueante

- Para primitivas não bloqueantes, parâmetro de retorno é um *handle* que pode ser usado para verificar o estado da chamada.
- Pode ser feito de duas formas:
  - Verificar através de um loop ou periodicamente, verificando se o handle foi "resolvido" (*posted*)
  - Disparar um *Wait* com *handles* como parâmetro. Geralmente bloqueia (blocking wait) até que um dos handles seja resolvido (*posted).*

Bloqueante / não bloqueante

- Primitiva send não bloqueante

Send(X, destino, handle_k);

...

...

Wait(handle_1, handle_2, ..., handle_k, ..., handle_m);

- Se quando *Wait* é invocado o processamento da primitiva já completou, o *Wait* retorna imediatamente. Caso contrário, *Wait* bloqueia e aguarda um sinal para "acordar".
- Quando a primitiva completa, o subsistema de software de comunicação seta o valor de handle_k e "acorda" qualquer processo com um *Wait* bloqueado por esse *handle.*

Bloqueante / não bloqueante

- Quatro versões da primitiva send:
  - *Síncrona bloqueante*
  - *Síncrona não bloqueante*
  - *Assíncrona bloqueante*
  - *Assíncrona não bloqueante*
- Duas versões da primitiva receive:
  - Síncrona bloqueante
  - Síncrona não bloqueante

Bloqueante / não bloqueante

- Send síncrono bloqueante
  - Dado é copiado do buffer do usuário para o buffer do kernel e enviado pela rede.
  - Depois que o dado é copiado no buffer do destinatário e uma chamada *receive* foi feita, uma confirmação enviada de volta ao remetente faz com que o controle retorne ao processo que invocou a primitiva *send,* completando-a.

- Fig 14(a)

Bloqueante / não bloqueante

- Send síncrono não bloqueante
  - Controle retorna ao invocador assim que *inicia-se* cópia dos dados para o kernel.
  - Parâmetro na chamada não bloqueante recebe o handle para posterior verificação do estado do *send* síncrono.
  - Notificação é postada após a confirmação retornar do destinatário.
  - Processo de usuário pode verificar se *send* síncrono não bloqueante completou através de testes sobre o handle ou utilizando operações *Wait* em tal handle.

  - Fig 14(b)

Bloqueante / não bloqueante

- Send assíncrono bloqueante
  - Processo de usuário invoca send
  - Fica bloqueado até dados serem copiados ao buffer do kernel
  - Em versão sem buffer: fica bloqueado até o dado ser copiado para a rede.

  - Fig 14(c).

Bloqueante / não bloqueante

- Send assíncrono não bloqueante
  - Processo de usuário invoca *Send*.
  - Fica bloqueado até *iniciar-se* a cópia dos dados para o buffer do kernel (ou rede, se não utiliza buffer).
  - Controle retorna ao processo assim que transferência é iniciada; handle é configurado.
  - Processo de usuário pode verificar se *send* assíncrono não bloqueante completou através de testes sobre o handle ou utilizando operações *Wait* em tal handle.
  - *Send* assíncrono termina quando os dados foram copiados para fora do buffer do usuário.
  - Verificação de término pode ser necessária se usuário quer reutilizar buffer.

  - Fig 14(d).

Bloqueante / não bloqueante

- Receive bloqueante
  - Chamada *receive* bloqueia até que dados sejam recebidos e escritos no buffer do usuário.
  - Controle é retornado ao processo do usuário.

- Fig 14(a).

Bloqueante / não bloqueante

- Receive não bloqueante
  - Chamada *receive* é registrada pelo kernel e retorna um handle para verificação posterior do término da operação de *receive* não bloqueante.
  - Handle é resolvido (posted) após o recebimento dos dados, que são copiados para o buffer especificado pelo usuário.
  - Processo do usuário pode verificar término do *receive* não bloqueante invocando operação de *Wait* sobre o handle.
    - Se o dado já chegou quando a chamada é feita, estará pendente num buffer do kernel e precisa ser copiada para o buffer do usuário.
  - Fig 14(b).

Características

- Send síncrono: mais fácil de usar da perspectiva do programador
  - Handshake entre *send* e *receive* faz comunicação parecer “instantânea”
  - Simplifica a lógica do programa, mas pode diminuir eficiência do processo remetente (remetente tem que esperar).
  - *Receive* pode não ser chamado até muito depois dos dados terem chegado ao buffer receptor; *send* fica bloqueado até *receive* ser chamado.

Características

- *Send* não bloqueante assíncrono é útil para dados maiores.
  - Permite que outras operações sejam feitas concorrentemente com a operação de *send*.
- *Send* não bloqueante síncrono evita atrasos, principalmente quando *receive* ainda não foi invocado no destino.
- *Receive* não bloquente é útil para receber dados maiores ou quando o *send* ainda não foi invocado.
- Chamadas não bloqueantes oneram o programador, que precisa cuidar do término das operações.

Sincronismo de processador

- Outro nível de sincronismo, em oposição ao sincronismo de primitivas de comunicação
- Sincronismo de processadores:
  - Processadores executam em passos “travados” com seus relógios sincronizados
- Não alcançável em sistemas distribuídos.
- Em sistemas distribuídos, sincronização pode ser implementada em nível mais alto
  - Sincronização em “passos”
  - Alguma forma de barreira de sincronização para garantir que um processador não inicie o próximo passo em um código até que todos os processadores tenham completado os passos anteriores que lhes foram atribuídos.

Sincronismo de execução

- Outra classificação no nível de sincronismo
- Execução assíncrona é uma execução onde:
  - Não há sincronismo de processador
  - Não há limite para divergência de relógios
  - Atraso de mensagens: finito mas ilimitado
  - Sem limite de tempo para cada processo executar um passo
  - Fig 16

Sincronismo de execução

- Execução síncrona
  - Processadores sincronizados
  - Divergência de relógios limitada
  - Envio + entrega de mensagens: ocorre em um passo lógico
  - Há tempo limite para um processo executar um passo
  - Fig 17

Sincronismo de execução

- É mais fácil projetar e verificar algoritmos assumindo execução síncrona
  - Natureza coordenada da execução facilita essas tarefas
- Entretanto, na prática é difícil obter um sistema completamente síncrono e ter mensagens entregues dentro de um limite de tempo determinado.
  - Sincronismo precisa ser simulado
  - Envolve atrasar ou bloquear processos por algum tempo
- Então, sincronismo é uma abstração que pode ser fornecida aos programas.
- Quanto menor o número de passos, ou sincronizações, dos processadores, menores são atrasos e custos.

Sincronismo de execução

- Idealmente, programas querem que processos executem uma série de instruções em cada passo (ou fase) de forma assíncrona e, após cada passo/fase todos os processos devem ser sincronizados e todas as mensagens enviadas devem ser entregues.
  - Entendimento padrão de execução síncrona.

Emulação

- Sistema assíncrono sobre um sistema síncrono (A→S)
  - Sistema síncrono: caso especial do assíncrono
  - Programa escrito para sistema assíncrono pode ser emulado trivialmente em um sistema síncrono onde toda comunicação termina dentro de uma mesma fase na qual ela foi iniciada
- Sistema síncrono sobre um sistema assíncrono (S→A)
  - Programa escrito para sistema síncrono pode ser emulado em sistema assíncrono usando uma ferramenta chamada sincronizador.

Emulação

- Vimos que memória compartilhada pode ser emulada em um sistema de troca de mensagens e vice-versa.
- Quatro classes de programas.
  - Assíncrono, troca de mensagens (AMP)
  - Síncrono, troca de mensagens (SMP)
  - Assíncrono, memória compartilhada (ASM)
  - Síncrono, memória compartilhada (SSM)
- Se um sistema A pode ser emulado por um sistema B, e se um problema não pode ser resolvido em B, também não pode ser em A.
- Se pode ser resolvido em A, pode ser resolvido em B.
- Fig 18

Emulação

- As quatro classes oferecem “computabilidade” (i.e., que problemas podem e não podem ser resolvidos) equivalente em sistemas livres de falhas.
- Em sistemas suscetíveis a falhas, sistemas síncronos oferecem maior “computabilidade” que assíncronos.

Desafios

Desafios

- Primórdios - 1970/ARPANET
  - Acesso a dados remotos sob falhas
  - Projeto de sistemas de arquivos
  - Projeto de estrutura de diretórios
  - Outros requisitos e problemas surgiram com o passar do tempo

- Relacionados a sistemas
- Relacionados a algoritmos
- Relacionados a tecnologias/aplicações recentes

Desafios - Sistema

- Comunicação: mecanismos apropriados para comunicação entre processos.
- Processos: gerência de processos e threads em clientes/ servidores; migração de código; software e agentes móveis.
- Nomenclatura: esquemas de nomenclatura robustos para localização de recursos e processos transparentemente, em especial para sistemas móveis.
- Sincronização: coordenação entre processos é essencial. Outras formas de exclusão mútua, sincronização de relógio, relógios lógicos, algoritmos para manutenção global de estado.

Desafios - Sistema

- Armazenamento e acesso a dados: esquemas de armazenamento, acesso rápido e escalável a dados na rede. Projeto de sistemas de arquivo direcionados a sistemas distribuídos.
- Consistência e replicação: evitar gargalos; fornecer acesso rápido a dados e escalabilidade. Gerência de réplicas e consistência.
- Tolerância a falhas: manter funcionamento eficiente mesmo na presença de falhas de enlace, processos ou nós. Resiliência de processos, comunicação confiável, checkpointing e recuperação, detecção de falhas.
- Segurança

Desafios - Sistema

- Transparência em diversos níveis: acesso, localização, migração, relocação, replicação, concorrência, falha.
- Escalabilidade e modularidade: algoritmos, dados e serviços tanto distribuídos quanto possível. Técnicas como replicação, caching e gerência de cache, e processamento assíncrono ajudam a alcançar escalabilidade.

Desafios – Algorítmicos

- Modelos de execução e arcabouços: especificações formais de modelos e conjuntos de ferramentas para raciocínio, projeto e análise de programas distribuídos.
- Algoritmos em grafos dinâmicos: algoritmos importantes para comunicação, disseminação de dados, localização de objetos, busca de objetos. Topologia e conteúdo dinâmicos, algoritmos influenciam latência, tráfego e congestionamento.

Desafios – Algorítmicos

- Comunicação em grupo, multicast, entrega de mensagens ordenadas: algoritmos para comunicação e gerência eficiente de grupos dinâmicos sob falhas.
- Monitoramento: algoritmos online para monitorar e mapear eventos e tomar ações.
- Ferramentas para desenvolvimento e teste de programas distribuídos.
- Depuração de programas distribuídos: desafio devido à concorrência; mecanismos e ferramentas são necessários.
- Algoritmos para gerência e replicação de dados, atualização de réplicas e cópias em cache. Alocação de cópias no sistema.

Desafios – Algorítmicos

- WWW – cache, busca, escalonamento: prefetching/padrões de acesso, redução de latência de acesso, busca de objetos, distribuição de carga.
- Abstrações de memória compartilhada.
- Confiabilidade e tolerância a falhas.
- Balanceamento de carga: aumentar vazão e diminuir latência. Migração de dados, migração de computação, escalonamento distribuído.
- Escalonamento em tempo real: aplicações sensíveis ao tempo. Problema se visão global do sistema não está disponível. Difícil prever atrasos de mensagens.

Desafios – Algorítmicos

- Desempenho: alta vazão pode não ser o objetivo primário de um sistema distribuído, mas desempenho é importante.
  - Métricas: identificação de métricas apropriadas para medir desempenho teórico e prático (medidas de complexidade versus medida com parâmetros/ estatísticas reais).
  - Ferramentas e métodos de medição: metodologias apropriadas e precisas para obtenção de dados confiáveis para as métricas definidas.

Desafios - Aplicações e avanços recentes

- Sistemas móveis: meio de broadcast compartilhado; problemas de alcance e interferência. Roteamento, gerência de localidade, alocação de canais, estimativa de posição, são problemas a serem tratados. Estação-base versus ad-hoc.
- Redes sensores: monitorar eventos distribuídos / streaming de eventos. Economia de energia na comunicação/middleware.
- Computação ubíqua e internet das coisas: auto-organização, recursos limitados; inteligência no núcleo da rede?
- Peer-to-peer: mecanismos de armazenamento persistente, busca e obtenção escalável, privacidade.

Desafios - Aplicações e avanços recentes

- Publish-subscribe, distribuição de conteúdo e multimídia: filtros de informação dinâmicos. Mecanismos eficientes para distribuição aos interessados e de inscrição de interessados. Mecanismos de agregação de informação e de distribuição de dados com grande volume.
- Grades computacionais: ciclos de CPUs ociosas; escalonamento e garantias de tempo real; predição de desempenho; segurança.
- Nuvens computacionais: virtualização; escalonamento / custos monetários; modelos econômicos; segurança.
- Segurança: soluções escaláveis para confidencialidade, autenticação e disponibilidade sob ações maliciosas.