Sistemas Distribuídos - INF 2545 - 2014.1

Questão sobre o servidor RPC.

Aluno: Rogério Carvalho Schneider <stockrt@gmail.com>

Professora: Noemi Rodriguez <noemi@inf.puc-rio.br>

Questão

É possível, na sua implementação de servidor que mantém conexões, que um servidor "jogue fora" uma conexão estabelecida com um novo cliente antes de chegar a tratar qualquer requisição desse cliente? Considere um cenário com alta concorrência de clientes. Por favor explique sua resposta.

Resposta

Antes de dar o parecer sobre a implementação de servidor RPC apresentada em aula, vamos analisar alguns cenários possíveis de paradigmas de programação para servidores.

Cenários a serem considerados

1. Servidor sem concorrência (apenas um processo atende requisições)

Apenas um servidor atende a todos os clientes, portanto não há concorrência no lado do servidor. O tratamento de cada conexão (*accept*) e de cada diálogo (*send*/*receive*) do protocolo é feito de forma sequencial.

Algumas formas possíveis de se implementar este comportamento são:

1.1) *Servidor consome a próxima conexão na fila* (a fila é formada no *backlog* do sistema operacional e tem um limite configurável durante o *bind*/*listen*). A conexão do novo cliente é adicionada ao *pool* de conexões do servidor, logo em seguida o diálogo do protocolo começa e a requisição de cada cliente que tem atividade no *pool* é atendida, um cliente por vez. Ao final do tratamento, a próxima conexão é consumida do *backlog*, expulsando ou não clientes já conectados (porém, já atendidos para cada diálogo reconhecido pelo servidor).

```
-- servers
for server in server_num -- multi servant
  servers += server:bind() -- one port per server
-- loop forever
for server in select(servers) -- check for client arrival activity
  client_pool += server:accept() -- pop client from kernel backlog, one at a time
  for client in select(client_pool) -- check for client request activity
    client:receive() -- protocol dialog for each client, one at a time
    client:send()
    ...
```

Neste cenário, uma nova conexão somente é atendida após o servidor e o cliente terem a chance de completar uma rodada de diálogo - chance não significa que de fato houve diálogo. Existe, contudo, a possibilidade de o cliente ser expulso do *pool* de conexões antes de ter qualquer requisição de diálogo tratada caso o cliente, após ganhar vez no *pool*, não envie uma requisição de diálogo em tempo. Um cliente conectado ao servidor e presente no *pool* de conexões será chutado por novos clientes mesmo sem ter feito o primeiro diálogo, caso seja um cliente muito lento. Uma sessão interativa, com telnet por exemplo, poderia ser considerada um cliente muito lento no sentido de que pode não enviar um comando *send* em tempo para ser atendido antes que vários outros clientes, em um *loop* automatizado, o façam ser expulso do *pool*.

Contudo, é importante observar que, uma vez iniciada a conversa com *send*/*receive* entre cliente e servidor, não há chance deste diálogo ser interrompido por novas conexões no servidor. Durante a conversa, o servidor não atende a outros clientes de nenhuma forma, e a sequência de *send*/*receive*, uma vez iniciada, irá seguir até o fim de acordo com o protocolo.

Um diálogo adicional, de um cliente já conectado, que deseja novamente conversar com o servidor, só será atendido após o término do diálogo corrente com o cliente atual do servidor. Isto garante que diálogos reconhecidos pelo servidor não serão interrompidos nem por novas conexões, nem por conexões reaproveitadas. Um diálogo reconhecido se caracteriza pelo primeiro *receive* completado pelo servidor, quando o mesmo inicia o tratamento de uma requisição de cliente.

Outra situação em que o cliente poderia sofrer negação de serviço é quando o *backlog* do sistema operacional estivesse cheio, sem capacidade de enfileirar uma nova requisição de conexão.

Um ponto negativo neste modo de funcionamento é que se um cliente, após ter iniciado seu diálogo com o servidor, se mostrar muito lento, irá atrasar todo o *loop* de tratamento, gerando contenção para todos os clientes que aguardam o fim da conversa para serem atendidos pelo servidor.

Este modo de funcionamento é síncrono e sequencial.

1.2) *Servidor consome todas as conexões da fila*. Após consumir o *backlog*, o servidor inicia o diálogo com os clientes do seu *pool* que tem algo a dizer (que fizeram *send* em tempo para esta rodada).

```
-- servers
for server in server_num -- multi servant
  servers += server:bind() -- one port per server
-- loop forever
for server in select(servers) -- check for client arrival activity
  loop client_pool += server:accept() -- pop all clients from kernel backlog
  for client in select(client_pool) -- check for client request activity
    client:receive() -- protocol dialog for each client, one at a time
    client:send()
    ...
```

Neste caso, como existe um *loop* de *accept*, é possível que um cliente tenha a sua conexão removida do *backlog* e, sem ter seu diálogo sequer iniciado, seja desconectado. Suponha que o tamanho do *pool* de conexões do servidor seja 3, e 10 clientes iniciem conexão ao mesmo tempo, formando um *backlog* de tamanho 10. O servidor iria consumir as 10 conexões do *backlog*, iria adicionar em seu *client_pool* as primeiras 3 conexões e logo em seguida começaria a descartar a conexão mais antiga para atender a próxima do *backlog*, descartando um total de 7 conexões sem sequer ter a chance de ouvir o que elas tinham a dizer.

Nessa situação, a perda está evidente no *overhead* gerado na reconexão de clientes, que pode acontecer sem necessariamente terem a oportunidade de ter dialogado via protocolo de aplicação (sem diálogo) com o servidor. Para o cliente, a desconexão antes de poder fazer o primeiro *send* é equivalente ao cenário de ser removido do *pool* antes de poder iniciar o n-ésimo diálogo, em conexão persistente. Na prática, para o cliente, ter a sua conexão com o servidor fechada antes do primeiro ou antes de qualquer outro diálogo deveria ter o mesmo efeito, o de reconectar para tentar iniciar o próximo diálogo numa nova conexão.

Nos casos 1.1 e 1.2 acima, como a comunicação é bloqueante e o tratamento é sequencial por apenas um servidor, não há concorrência ou qualquer maneira de alternância no tratamento de conexões ou diálogo dos clientes. Desta maneira, apenas o *loop* de *accept* deslocado do item 1.2 parece oferecer trabalho extra quase que garantidamente em alta concorrência, tanto para o servidor quanto para o cliente. Da mesma maneira, caso haja um cliente muito lento em iniciar diálogo após ter sido inserido no *pool*, também nos dois casos 1.1 e 1.2, há a possibilidade de ser realizado trabalho extra de reconexão. Nenhuma das duas sugestões possibilita, entretanto, que um cliente que tenha seu diálogo já reconhecido seja interrompido por uma nova conexão de outro cliente, nem pela velocidade do diálogo dos outros clientes, nem pela sua própria velocidade, nem pelo volume de clientes. Contudo, clientes muito lentos em iniciar seu diálogo

podem sim ser desconectados antes de serem atendidos, caso o volume e velocidade dos outros clientes seja grande.

2. Servidor com concorrência (mais de um processo atende requisições)

Neste cenário, mais de um processo atende a clientes ao mesmo tempo, gerando concorrência. O tratamento de cada conexão (*accept*) e de cada diálogo (*send*/*receive*) do protocolo é feito de forma sequencial em cada processo, mas existem vários processos, chamados *workers*, que atendem a diferentes clientes em paralelo.

2.1) *Pool de conexões gerenciado de forma local*. Cada servidor tem o seu *pool* de conexões e atua de forma semelhante a um servidor não concorrente, com tratamento sequencial como no proposto nos itens 1.1 e 1.2 do cenário anterior. A diferença é que neste caso cada servidor disputa o consumo do *backlog* para a composição de seu *pool* local.

```
-- servers (do bind/listen)
for master_server in server_num -- multi servant
  master_servers += master_server:bind() -- one port per server
-- workers (prefork)
for worker in woker_num -- master's workers
  worker = fork()
  if worker:child? -- is a worker, else loop next
    -- loop forever
    -- worker code (do all select, do accept, own pool, handle protocol)
    for server in select(master_servers) -- check for client arrival activity
      client_pool += server:accept() -- pop client from kernel backlog, one at a time
      for client in select(client_pool) -- check for client request activity
        client:receive() -- protocol dialog for each client, one at a time
        client:send()
          ...
```

A execução do código do *worker* é similar ao passo sequencial do item 1.1 do cenário não concorrente, no qual o *accept* é feito para um cliente de cada vez, garantindo que o *pool* local somente descarte conexões que já tiveram pelo menos a chance de ter a sua provável requisição tratada naquela rodada. O sistema operacional garante que a chamada de *accept* seja atômica, evitando que dois *workers* consumam do *backlog* a mesma conexão de cliente.

Como o *pool* de conexões é local para cada *worker*, não há concorrência na manipulação das conexões, que acabam sendo tratadas uma de cada vez em seu *worker*.

Do ponto de vista interno do *worker*, o comportamento e defeitos esperados são os mesmos do item 1.1. Do ponto de vista externo, a concorrência gerada no consumo do *backlog* não impacta

no gerenciamento dos *pools* locais de conexões nem no tratamento local do diálogo de protocolo de cada *worker* com seus clientes.

Interessante observar que nesta implementação os *workers* são pré-instanciados (*prefork*) e permanecem em *loop* atendendo o tempo todo ao fluxo de clientes.

2.2) *Pool de conexões gerenciado de forma global*. O processo *master* é responsável tanto pelo *bind*/*listen* quanto pelos *accepts* e o *pool* de conexões agora é o mesmo entre os *workers* de um mesmo servidor.

```
-- servers (do bind/listen, do server select, do accept, own pool)
for master_server in server_num -- multi servant
  master_servers += master_server:bind() -- one port per server
-- loop forever
for master_server in select(master_servers) -- check for client arrival activity
  client_pool += master_server:accept() -- pop client from kernel backlog, one at a time
  -- workers (forked on demand)
  worker = fork()
  if worker:child? -- is a worker, else loop next
    -- worker code (do client select, handle protocol)
    for client in select(client_pool) -- check for client request activity
      client:receive() -- protocol dialog for each client, one at a time
      client:send()
      …
```

Neste caso, a pressão no consumo do *backlog*, durante o *accept*, não é mais diluída nos *workers*, ficando concentrada no processo *master*. Como nesta implementação o *master* concentra todos os *accepts*, ele precisa fazer *fork* a cada nova conexão com a finalidade de repassar a um novo *worker* o tratamento do diálogo com o *socket* de cliente, e isto também gera mais pressão no processo *master*. Como os *workers* são instanciados sob demanda, também há um custo maior em performance, porém, há ganho em isolamento de memória entre diferentes requisições (evita *memory-leaks*, por exemplo).

No cenário particular apresentado neste item é possível identificar que, como nos outros cenários, um cliente muito lento em iniciar diálogo poderá ser removido do *pool* antes de conversar com o servidor, e ainda mais, neste cenário um cliente poderá ser expulso do *pool* de conexões no meio de um diálogo, situação inédita em nossa análise até agora.

Para que um cliente seja expulso no meio de um diálogo, basta que haja muita atividade em outros *workers*, de maneira que o *pool* de conexões compartilhado fosse descartando conexões mais antigas à medida que novos clientes surgissem. A conexão mais antiga, eventualmente, poderia ser a conexão de um cliente considerado lento em seu diálogo, que poderia ser expulso do *pool* sem qualquer aviso, quando um novo cliente fosse atendido. Este cenário é muito ruim

pois traz insegurança sobre em que pontos se pode contar com o atendimento do servidor e em que pontos se deve esperar que uma nova conexão pode precisar ser estabelecida. Como o servidor não mantém estado na conversação justamente por ser um *worker* instanciado sob demanda, não há nem como um cliente reconectar e continuar de parou a sua conversa. De qualquer maneira, este cenário é indesejado.

O último caso de cada cenário, 1.2 e 2.2, são na verdade *anti-patterns* no desenvolvimento de servidores. Estes cenários contém pequenos defeitos de projeto e serviram para ilustrar casos propensos a apresentar problema conforme o aumento de carga no sistema, que vem junto com o aumento do número de clientes, e conforme o perfil de uso da aplicação, se por clientes muito rápidos, muito lentos ou mistos.

3. Servidor responsivo (assíncrono e não bloqueante)

Neste último cenário, um servidor atende e trata conexões e requisições de forma assíncrona, intercalando o atendimento conforme haja algum tipo de atividade a ser tratada.

3.1) *Servidor com reator no loop principal*. Totalmente assíncrono e não bloqueante.

```
-- servers
for server in server_num -- multi servant
  servers += server:bind() -- one port per server
-- reactor (responsive loop forever)
when server in select(servers) -- when client arrival activity
  client_pool += server:accept() -- pop client from kernel backlog, one at a time
when client in select(client_pool) -- when client request activity
  when client:receive() -- when client say something
    handle_protocol -- each client has it's own context to keep track of the conversation
  async client:send()
  …
```

O *loop* do reator, que é o *loop* principal, irá iterar até que haja atividade em qualquer uma das chamadas *when*, seja atividade de novo cliente, seja durante atividade de diálogo de um cliente já conectado. Como não há preferência na ordem de execução entre tratamento de novas conexões e de atividade de protocolo, qualquer uma que ocorrer primeiro será tratada pelo reator.

Um evento de comunicação pode ser atendido pela chamada *when* quando não houver atividade a ser tratada pela CPU (quando não está executando o método local solicitado pelo cliente, via RPC, por exemplo). Claro que, durante o uso da CPU do servidor, um cliente pode estar preparando e enviando dados, e tão logo a CPU do servidor esteja liberada o reator irá perceber que existe evento de rede para tratar, e irá disparar a chamada *when* adequada.

Os eventos de comunicação se caracterizam pela presença de dados para serem consumidos em um canal. Quando há o que ler em um *socket* de cliente ou servidor, o evento é disparado, desta forma o processo não fica bloqueado durante a transmissão dos dados pela rede. Da mesma forma, quando um envio é feito, o *loop* principal de eventos não fica preso esperando que a outra ponta consuma os dados postados. Este conjunto de operações caracteriza um servidor assíncrono e de comunicação não bloqueante.

O ponto a observar neste paradigma é que durante o tratamento do diálogo, se houver possibilidade, o reator irá tentar atender a chegada de novos clientes nos intervalos em que não há comunicação entre clientes já conectados e o servidor. O problema aparece neste caso, em que fica evidente que se o tratamento assíncrono for feito desta forma, é muito provável que alguns clientes sejam expulsos do *pool* antes de terminar seu diálogo já iniciado com o servidor, isto devido ao fato de que novos clientes continuam chegando e sendo atendidos.

Para deixar mais complexa a arquitetura de um servidor, podemos misturar servidores assíncronos com servidores concorrentes. Neste caso os *workers* seriam assíncronos e não bloqueantes. Na prática apenas a implementação seria mais complexa, pois a análise isolada de um *worker* assíncrono se assemelha à análise de um servidor simples assíncrono e apresenta os mesmos problemas.

A implementação RPC apresentada em aula

Por fim, respondendo a pergunta sobre a implementação de servidor apresentada em aula para o trabalho de RPC, luarpc, o paradigma adotado se assemelha ao do item 1.2, portanto sim, é possível que o servidor descarte conexões de clientes antes mesmo de tratar qualquer requisição deles.

Somente após este estudo e durante alguns testes é que ficou evidente que há este comportamento indesejado no servidor implementado. Certamente, em um cenário de alta concorrência de clientes, este servidor iria proporcionar retrabalho, fazendo com que clientes já conectados não fossem atendidos, gerando uma nova conexão desnecessária e custosa em termos de performance.

Também seria possível observar que um cliente lento, durante o atendimento de sua requisição, poderia atrasar o atendimento de todos os outros clientes. Uma vez que o servidor é simples, não concorrente, ele somente processa o diálogo de um cliente por vez. Para impedir que um cliente ocupe o servidor indefinidamente, foram utilizados *timeouts* na configuração dos *sockets*.

Para melhorar a leitura, o código abaixo foi reduzido e algumas chamadas foram suprimidas.

```
-- luarpc server loop
local server_accept_ready_list, _, err = socket.select({servant.server}, nil, 0)
  for _, server in pairs(server_accept_ready_list) do
    local client = server:accept()
    table.insert(servant.client_list, client)
    while #servant.client_list > servant.pool_size do
      old_client = table.remove(servant.client_list, 1)
      old_client:close()
-- server/client dialog
local client_recv_ready_list, _, err = socket.select(servant.client_list, nil, 0)
  for _, client in pairs(client_recv_ready_list) do
    client:receive() -- protocol dialog for each client, one at a time
    client:send()
    …
```

No trecho em verde é possível verificar que os clientes são aceitos para novas conexões, todos os clientes que tenham até aquele momento pedido para se conectar, e isto se assemelha em muito ao *loop* de *accept* do item 1.2. No trecho em vermelho, os mesmos clientes já aceitos, caso a demanda seja alta, podem vir a ser descartados do *pool* de conexões para que os clientes restantes no *backlog* sejam consumidos. Somente depois destas iterações é que os clientes que sobraram no *pool* de conexões começarão a ser atendidos, um a um, sem chance de terem seu diálogo interrompido mas com a possibilidade de causar lentidão no atendimentos uns dos outros e empilhamento de novas conexões no *backlog*.

Considerações finais

Durante a escolha da implementação de um servidor é muito importante observar qual o perfil de uso esperado para a aplicação, tomar cuidado para não cair em armadilhas comuns de erro de projeto, observar facilidade de depuração de código, manutenção e performance e então optar pelo paradigma que melhor se aplica ao caso de uso em questão.

Nem sempre é possível determinar o número de clientes esperados, a frequência com a qual irão interagir com o servidor e nem a sua velocidade, por isto é importante preparar o código do servidor para ser bem comportado e educado com os clientes sempre que possível. Idealmente, um servidor não deveria deixar um volume alto de clientes promover retrabalho nem para si, nem para os clientes já em atendimento, não deveria permitir a frequência de reconexão de clientes impactar nem diálogos já iniciados, nem conexões já estabelecidas mas não servidas e, por fim, não deveria deixar um cliente lento causar demora no atendimento dos demais clientes. Para se proteger destes casos é preciso avaliar, de acordo com a aplicação, qual o paradigma de programação de servidor mais adequado.

As implementações sugeridas neste trabalho não fornecem proteção para todos os casos problemáticos indicados, porém algumas medidas podem ser aplicadas de acordo com cada implementação. Por exemplo, o uso de estruturas mais complexas para gerenciamento de *pool* de conexões, com regras e *flags* definidas para permitir ou não a remoção de um cliente do *pool*, com penalidades definidas para clientes lentos, com o uso de algoritmos de expurgo e com tamanho variável de *pool*.

Um caso interessante, por exemplo, seria definir saltos condicionais para evitar um *accept* em cliente novo se o *pool* já estiver cheio, com clientes em meio a diálogos e que não estejam se comportando de forma lenta o suficiente para serem considerados para expurgo do *pool*, ou viabilizar o *accept*, incrementando o tamanho do *pool* caso os indicadores de performance do servidor estejam dentro de limites aceitáveis, criando um *pool* dinâmico. Esta opção mais elaborada de inserção no *pool* é bem mais educada por parte do servidor, deixando que um cliente novo tenha o acesso negado ao invés de desconectar um cliente que já está em meio a um atendimento e que parece estar se comportando bem, ou aceitando este novo cliente caso seja possível expandir o *pool* um pouco mais. A complexidade de tratamentos, necessidade de *locks* em estruturas compartilhadas e a manutenção de um servidor com tais recursos será maior do que em um servidor mais simples. Novamente, a opção por um ou outro modelo deve ser feita em tempo de projeto, seguindo os requisitos necessários para que o servidor implementado atenda de maneira satisfatória ao propósito para o qual foi criado.