Simple Polynomial Solver (SiPoSo)

April 3, 2022

1 Descrição

Alguns detalhes de implementação de mais baixo nível como o funcionamento de *links* e para o que podem funcionar (e.g transações de base de dados), bem como o funcionamento de *trap* de erros, o tipo de sinais de erros e o funcionamento de eventos para receber mensagens não será muito abordado.

Fazendo uma descrição da arquitetura que seguimos para modelar a interação dos vários componentes. Existe uma máquina designada de *"Supervisor"* referido daqui para a frente como S. Esta máquina é o orquestrador dos servidores que vão servir os pedidos dos clientes. S é implementado como uma *"interface"* dada pelo modulo *supervisor* do *Erlang*. Este modulo fornece as especificações do comportamento esperado de um supervisor. É esperado ao supervisor que ele de *restart* aos seus "filhos" sabendo a especificação dos mesmos e segundo uma estratégia definida. Isto permite ter uma forma automatizada e resiliente para prevenir que erros inesperados, mas previsíveis possam ser contornados de forma automática, ao empregar a técnica de zero tolerância e dar *restart* por completo ao "filho" envolvido.

Indo primeiramente a alguns detalhes sobre o comportamento especificado a S. A estratégia de *restart* utilizada foi o *one_for_one*. Assim, cada "filho" que terminar e tiver de ser reiniciado, não irá afetar os outros. Para definir a intensidade de *restarts*, S foi configurado para se em dentro de 60 segundos houver 3 *restarts* acontecerem, S entende isto como uma anomalia, prevenindo assim ciclos infinitos de *restart*, e desliga todos os seus "filhos" e termina-se a si próprio. O *"Manager"*, falado mais para a frente terá de resolver este acontecimento.

Detalhando agora um pouco sobre os "filhos" de S. Todos os "filhos" são registados no espaço interno de S de forma sequencial, e de forma global segundo um nome dado pelo *"Manager"*. Assim, é evitado problemas com *pids* alterarem-se, mantendo como "landmark" para cada "filho" a referência global gerida pelo próprio *Erlang*. Cada "filho" vai ser representado daqui para a frente por F, e implementa a "interface" dada pelo módulo *gen_server*. Deste modo, por cada n de F que tivermos podemos servir n clientes paralelamente. Ao dedicarmos um processo *gen_server* para cada cliente oferecemos de forma mais fácil a nível de implementação uma boa capacidade de computação, visto que cada F está preparado para poder ser "consciente" sobre a sua própria computação e poderá distribuir-la não só por outros processos paralelamente na mesma máquina, como em nós complemente diferentes em máquinas remotas. No entanto, estas funcionalidades não foram implementadas por, a nosso ver, saírem do objetivo do projeto. Apesar disso, foi implementado a delegação da computação dentro de F para outro processo para que este pode-se acumular pedidos na sua *queue* para fazer mais tarde. Para controlar esse processo de delegação da computação foi implementado um supervisor "do zero", muito simples, que apenas dá *restart* ao processo se este sair de forma anormal. Este processo é feito no máximo 10 vezes.

Passando agora a explicar a forma como os processos F são criados. Inicialmente, vamos ter 2 F criados quando S é inicializado. Estes dois processos F são especificados para serem permanentes, isto é nunca devem deixar de existir desde que o supervisor esteja em funcionamento. No caso de S ser terminado, este dará 2s para que estes dois F possam terminar por si próprio, fazendo se necessário alguma operação (e.g fechar ficheiros). Se ao final de 2s estes não fecharem, S força que estes terminem. Estes dois F iniciais são do tipo *worker*, tal como serão todos os possíveis processos deste género criados a mando do *"Manager"*.

Quanto ao *"Manager"*, referido daqui para a frente como M, como indiciado até agora, é responsável pela manutenção da arquitetura a funcionar. Este serve como uma espécie de *consensus server*, onde temos toda a informação sincronizada dos servidores disponíveis para os clientes. Através do manager podemos também criar novos processos F (mudando possivelmente os seus parâmetros). Estes novos servidores são supervisionados por S e estarão disponibilizados para que os clientes os possam usar. M tem também a capacidade de parar qualquer F, reiniciar qualquer F ou até mesmo terminar qualquer F. M pode também terminar S e dependendo das circunstâncias, isto é, da configuração física e lógica que estas máquinas sejam postas, dar inicio a S. De uma forma geral, a aquitetura encontra-se sumarizada na figura 1.

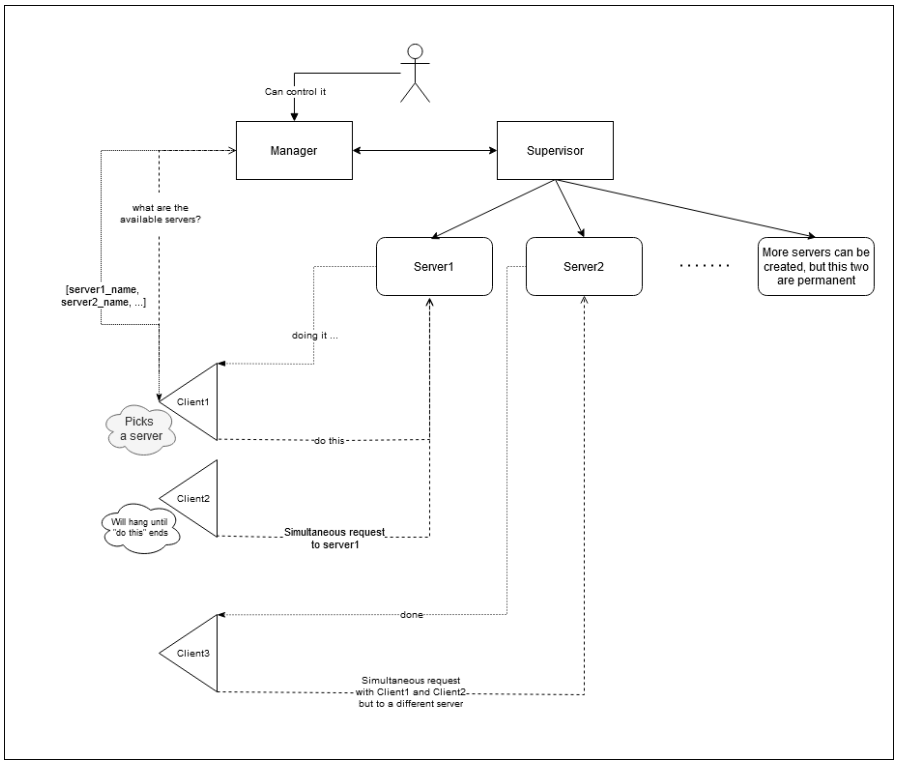

Figure 1: Vista geral da arquitetura desenvolvida.

Passando agora a uma breve explicação das funções usadas para calcular os resultados das operações com polinómios. Em Primeira instância, cada polinómio é validado pela função *check_integrity*. Isto garante que não existe nenhum erro na escrita do polinómio quando este é dado pelo cliente. Quanto há organização geral desta parte das funções matemáticas, isto é soma, subtração e multiplicação de polinómios, foi implementado uma função "regente", o *handler*. Esta função está encarregada de orquestrar a colocação do polinómio na forma correta após aplicar a função pedida (e.g soma), e de delegar a computação pedida à função correspondente.

Todas as funções são suportadas pela função *normalize*. Esta tem como objetivo simplificar, isto é juntar todos os termos que podem ser juntos. E assim sendo obter cada polinómio na forma mais compacta possível.

A função *sum* tira proveito total do conceito implementado pelo *normalize*. Ou seja, para somar, basta juntar ambos os polinómios (*append* nas listas), e usar a função *normalize* para que este seja simplificado. A função *sub* chama a função *sum*, pois subtrair o segundo é o mesmo que somar o complementar, isto é, negar todos os coeficientes.

A função que poderia ser mais complicada seria a *mult*, mas com o uso de listas em compreensão, tornou-se relativamente simples. Tirando proveito do funcionamento das listas em compreensão, foi somente necessário fazer uma função que dadas variáveis e expoente somasse os expoentes, obtendo assim o produto de ambos os termos em um termo único. Para este passo foi essencial a função *merge*. Esta poderá ser substituída por *lists:merge()* com uma função lambda, contudo e para efeitos de transparência, foi implementado uma versão nossa. Esta função junta duas listas ordenadas produzindo uma lista ordenada, onde os membros com o primeiro elemento igual somam os expoentes. Isto no fundo simboliza a soma de expoentes quando multiplicamos na aritmética elementar.

As funções *clean*, *clean_get_vars* e *clean_get_exp* são utilizadas para eliminar tanto variáveis com expoentes a 0 como termos com coeficientes 0, eliminado assim ruído desnecessário no polinómio resultante.

2 Problemas

Um detalhe que está implícito na descrição da aquitetura dada acima, mas que é importante reforçar, e que está descrito também na figura 1, é que se 2 clientes fizerem pedido ao mesmo tempo ao mesmo servidor, um deles terá de esperar que o outro receba a resposta do servidor para que depois seja atendido. Isto só é um problema pois é dada ao cliente a possibilidade de escolher o servidor. Ainda assim, mesmo que fosse dada a opção ao cliente de escolher um servidor, se existisse forma de este perceber o *load* atual de cada servidor, este poderia, a seu próprio risco escolher qualquer servidor, sujeitando-se a possiveis tempos de espera em servidores mais carregados. A implementação de mecanismos de *load balancing*, bem como a possibilidade de os clientes poderem saber o estado do servidor não foram implementados por (1) não serem triviais; (2) saírem do objectivo deste projeto.

Outro possível problema é que se M a qualquer momento deixar de funcionar, a única forma de os clientes interagirem com o serviço será adivinharem o nome registado globalmente de um dos *servers* disponíveis atualmente.

3 Como Testar

Existe um ficheiro com testes unitários sobre as funções de matemática principais. Ao compilar, basta correr também o modulo de testes unitários para garantir a integridade das funções. Ainda nesse modulo de testes unitários existe testes para garantir a conectividade de prontidão de S e de M quando falamos de comunicação entre si. No entanto, estes testes, não podem ser concluídos com sucesso sem os servidores estarem ligados. Teoricamente é possível, com transformações triviais transformar o código de forma a que ele seja testado localmente. Mas, uma vez que vai haver uma apresentação, deixamos para essa altura a demonstração com os servidores ligados. Se o professor quiser fazer testes pode nos contactar para ligarmos os servidores, ou pode até, se tiver disponibilidade converter o código para correr em modo local.

4 Nota

O código foi construído de forma a que acrescentar novas funções aos servidores seja banal. Apenas precisamos de acrescentar a função (uma qualquer função genérica, para uma funcionalidade qualquer) e definir-la no *handler()* e acrescentar-la ao cliente. Depois de a função ter a lógica implementada, do lado dos servidores, apenas temos de trocar o ficheiro que define a concretização das funções, sem alterar a estrutura e integridade dos servidores, podendo estes continuar a funcionar de forma ininterrupta. A escala de quantidade de servidores geridos por S bem como a hierarquia de supervisor-supervisionado, pode, teoricamente, embora não se tenha testado, ser alterada em tempo real.