Relatório do Trabalho Prático 3

14 de junho de 2018

Aluno: Ramon Melo
Professor: Daniel Figueiredo
`https://github.com/ramonduarte/sd3`

1 Objetivo

Construir um sistema distribuído cujo mecanismo de ordenação total de eventos seja baseado no algoritmo *Totally Ordered Multicast*, utilizando o relógio lógico de Lamport como ordenador.

2 Decisões de Projeto

O código utilizado para este trabalho, bem como os logs dos estudos de caso, as imagens, o relatório e a apresentação, estão disponíveis publicamente no caminho `https://github.com/ramonduarte/sd3`.

Ao contrário dos trabalhos anteriores, a ordenação total de eventos distribuídos ocorre numa camada de abstração significativamente acima do hardware, de forma que implementações de baixo nível e acesso direto ao metal não compõem mais o conjunto de ferramentas desejável ao desenvolvimento da aplicação. Pelo contrário, bibliotecas de *sockets*, processos e *user-level threads* estão disponíveis em praticamente todas as linguagens de programação. Desta forma, deseja-se do ecossistema características como o suporte a construtos de programação orientada a objetos, tais como herança e polimorfismo; legibilidade do código; e agnosticismo em relação ao sistema operacional.

A linguagem escolhida para este trabalho foi Python 3.5, edição que é nativa à maioria esmagadora de sistemas operacionais baseados no padrão POSIX. Em especial, esta versão é significativamente mais eficiente que as baseadas em Python 2 - mais tradicionais - e traz nativamente bibliotecas

como a `multiprocessing` - *user-level threads* em concorrência, uma das poucas implementações que suporta o ecossistema Windows (Foundation (2017)).

Dada a natureza dos requisitos, que exige estruturas de dados razoavelmente similares mas autônomas, e ao prazo de entrega, que forçou a realização de *sprints* bastante curtos (em média, dois por semana), o paradigma de orientação a objetos foi uma escolha natural.

Conforme sugestão do enunciado, a comunicação interprocessual foi mantida através de *sockets* TCP, o que, na prática, implica a premissa de que a rede física subposta é confiável, ou seja, mensagens não são perdidas (e, portanto, não precisam ser reenviadas) e chegam convenientemente de acordo com a ordem em que foram enviadas (Kurose (2007)). Esta premissa pode ser considerada verdadeira para *sockets* locais, que não dependem de canais externos ao sistema operacional para completarem o percurso. Por esta razão, os estudos de caso foram realizados no mesmo computador, onde tal presunção pode ser verificada e mensurada.

É importante mencionar que, devido à especificação do trabalho, ferramentas de altíssimo nível de abstração foram propositalmente evitadas, visto que mascarariam funcionalidades requeridas pelo enunciado. Ainda assim, é relevante mencioná-las, para fins de completude desta seção. O módulo `socketserver` gerencia silenciosamente falhas de conexão em *sockets*, e seria a principal escolha se não houvesse especificada a necessidade de coordenar manualmente os *sockets*. Caso não houvesse a necessidade de utilizar uma *thread* cliente e outra servidora, o módulo `asyncio` seria uma escolha mais intuitiva. Se esta fosse uma aplicação comercial, muitas das funcionalidades já estariam implementadas no framework *Twisted*, direcionado a arquiteturas orientadas a eventos, como é o caso aqui (Labs (2018)).

Embora a linguagem preveja implementação agnóstica em relação aos sistemas operacionais, esta aplicação foi desenvolvida sob o sistema operacional Ubuntu 14.04 LTS 64-bit, onde a implementação é determinística (Foundation (2017)). Em especial, versões antigas de sistemas Windows podem exibir condições de corrida não-previsíveis.

O programa foi projetado para ser executado por um interpretador no modo otimizado (`python -O main.py [args]`). Embora o modo não-otimizado produza o mesmo resultado para fins práticos, há diversos gatilhos de depuração (`if __debug__`) que serão ativados. A avaliação pode se tornar cansativa ou tediosa devido à verbosidade das diretivas de depuração.

3 Implementação

Há cinco classes que serviram de fundação para o trabalho. O diagrama de classes está representado graficamente na Figura 1.

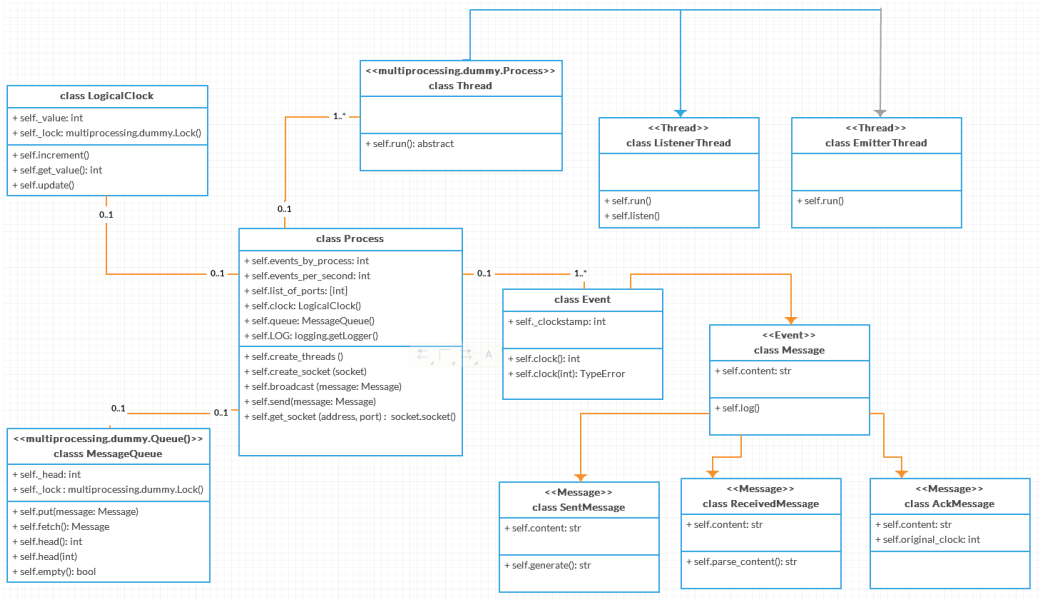

Figura 1: Diagrama de Classes da implementação.

A classe `Process` encapsula funcionalidades dos módulos `socket` (McMillan (2017)) e `multiprocessing`. É importante observar que este último expõe a mesma *API* tanto para processos, quanto para *threads*. Devido aos requerimentos do enunciado, foi utilizado somente o *namespace* `dummy`, que inicia *user-level threads* dentro do mesmo processo e permite o compartilhamento implícito de memória entre elas. Para cada `Process`, é aberto um total de 3 *threads*: uma cliente, uma servidora e uma principal, responsável pelo gerenciamento dos recursos do processo. O encerramento desta última automaticamente interrompe as demais e ativa a coleta de lixo e desalocação de recursos, incluindo os *sockets* previamente alocados.

O autor observou que, a despeito disto, eles permanecem abertos e ocupando o endereço por algum tempo, presumidamente por opção do sistema operacional (Kiehl (2015)). Caso isto ocorra, o programa acionará uma exceção `ConnectionRefusedError` e encerrará logo em seguida. Falhas capturadas pelo interpretador Python, incluindo o comando `CTRL+C`, provocarão, como reação, a tentativa de fechar todos os *sockets* antes de encerrar a execução. Falhas que escapam ao contexto do interpretador (como o sinal `SIGKILL`)

provocarão o encerramento abrupto, caso em que `ConnectionRefusedError`s são mais prováveis de ocorrer.

A classe `Thread` concentra os métodos executados concorrentemente. Em especial, o método `run()` foi construído desde o princípio de forma limitada ao padrão *thread-safe*. Herdam desta classe `ListenerThread` (orientada a serviços de servidor) e `EmitterThread` (orientada a serviços de cliente).

Os objetos que representam os eventos são derivados da classe `Event`. Particularmente, os que representam as mensagens derivam da classe `Message`. São eles: `SentMessage` (enviada pelo processo que a criou), `ReceivedMessage` (recebida pelo processo que a criou) e `AckMessage` (confirmação a ser enviada pelo processo que a criou). As mensagens são as responsáveis pela execução, visto que representam os eventos na abstração do mecanismo *Totally Ordered Multicast* enunciada a este trabalho. Portanto, é a classe `ReceivedMessage` a responsável por gravar pertinentemente as mensagens no disco uma vez que estejam aptas.

O relógio lógico de Lamport é representado pela classe `LogicalClock`, que encapsula cadeados (*locks*) para a manutenção do caráter *thread-safe* do método `Thread.run()`.

Por fim, as mensagens que aguardam execução são armazenadas num objeto `MessageQueue`, que encapsula uma fila do tipo *FIFO* (*First In, First Out*, "fila indiana") e cadeados para seu acesso concorrente.

Todos os objetos são pertencentes ao objeto `Process` que coordena a execução local do programa. Isto porque, para garantir o acesso implícito à memória compartilhada, o módulo `multiprocessing` exige que as estruturas de dados sejam referenciadas pela abstração do processo (e não pela das *threads*).

A escrita no disco é gerenciada pelo modo `logging`, que grava num arquivo sugerido pelo próprio usuário como comando.

Os argumentos passados ao arquivo `main.py` incluem o nome do arquivo do log onde os eventos serão gravados, o número n de mensagens enviadas por processo, a taxa λ de emissão de mensagens por segundo e o total p de processos abertos simultaneamente.

4 Estudos de Caso

Os estudos de caso foram executados para as combinações dos valores $n \in \{1, 2, 4, 8\}$, $\lambda \in \{100, 1000, 10000\}$ e $p \in \{1, 2, 10\}$. Foram usadas portas de numeração entre 8000 e 8007, por serem comumente livres na maioria dos sistemas. Os valores podem ser conferidos na Tabela 1. Os logs estão disponíveis no repositório onde o trabalho foi hospedado, na pasta `logs/`.

p	λ	tempo (s)	p	λ	tempo (s)	p	λ	tempo (s)
2	1	28.52374	2	1	286.246658	2	1	2998.55037
2	2	12.420049	2	2	148.901997	2	2	1494.091501
2	10	2.959135	2	10	29.346077	2	10	297.763825
4	1	59.568429	4	1	585.1391	4	1	5883.879098
4	2	29.813078	4	2	296.462348	4	2	2979.343469
4	10	5.086868	4	10	59.496107	4	10	600.332818
8	1	109.699103	8	1	1152.87353	8	1	11834.078567
8	2	55.250551	8	2	585.239865	8	2	5929.261745
8	10	10.990417	8	10	113.358163	8	10	1182.840992

Tabela 1: (a) $n = 100$; (b) $n = 1000$; (c) $n = 10000$.

A análise dos tempos corrobora a premissa de rede confiável, sobretudo se se considerar que os testes foram conduzidos em `localhost`. A complexidade da execução é trivialmente verificável como $O(n)$ e $O(\lambda)$. O número de processos teve impacto desprezível sobre o tempo total de execução.

Um olhar mais perspicaz sobre os logs, porém, revela que houve um número significativo de processos interrompidos antes do tempo. A maioria das falhas críticas parece apontar para a perda de conexão do *socket*, haja vista que o interpretador Python levanta a exceção `BrokenPipeError` nestas condições. Há, contudo, um conjunto de exceções que não foram capturadas, provocando o encerramento precoce dos processos. Como a falha só foi detectada às vésperas do prazo, não houve tempo hábil de depurá-la. A justificativa para tanto é o fato de só ser possível notar tal aspecto lendo os logs até o fim, que são arquivos cuja extensão desafia os limites da janela de atenção do cérebro humano.

5 Considerações Finais

Apesar de simples em essência, a implementação deste mecanismo foi cercada de desafios teóricos e práticos. O tempo total de desenvolvimento foi de 116 horas, divididas em 3 semanas.

A principal dificuldade, do ponto de vista teórico, foi a construção do diagrama de classes para o projeto. As escolhas que pareciam mais prováveis para uma comunicação entre dois processos provaram-se desastrosas quando expandidas.

Do ponto de vista prático, o principal desafio foi gerenciar a integridade dos *sockets*. A biblioteca nativa do Python 3.5 não possui um método confiável de checagem de integridade. De fato, a recomendação oficial é, justa-

mente, que se tente enviar mensagens quando desejado e que se capture a exceção `BrokenPipeError` quando levantada.

Ainda assim, o autor acredita que o projeto promoveu uma revisão abrangente e profunda da ordenação total de eventos em sistemas distribuídos. De fato, se refeito do princípio, haveria um esforço maior na etapa de manutenção da integridade dos *sockets*, visto que são, de longe, o maior gargalo desta implementação.

Referências

Foundation, P. S. (2017). Official 3.5.2 documentation: Process-based parallelism. `https://docs.python.org/3.5/library/multiprocessing.html`. Accessado em 14/06/2018.

Kiehl, C. (2015). Parallelism in one line. `http://chriskiehl.com/article/parallelism-in-one-line/`. Accessado em 14/06/2018.

Kurose, J. F. (2007). *Computer Networking: A Top-Down Approach*. Pearson.

Labs, T. M. (2018). Twisted: An event-driven networking engine written in python. `https://twistedmatrix.com/trac/`. Accessado em 14/06/2018.

McMillan, G. (2017). Official 3.5.2 documentation: Socket programming. `https://docs.python.org/3/howto/sockets.html`. Accessado em 14/06/2018.