Data:	13 de Junho de 2018
Aluno:	Ramon Melo
Professor:	Daniel Figueiredo

1 Objetivo

Construir um sistema distribuído cujo mecanismo de ordenação total de eventos seja baseado no algoritmo *Totally Ordered Multicast.*

2 Decisões de Projeto

Balance used	*#4*
Magnesium from sample bottle	*#1*

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

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 (**?**). 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*.

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 determinísticaFoundation (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.

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.

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` (mensagem enviada pelo processo que a criou), `ReceivedMessage`

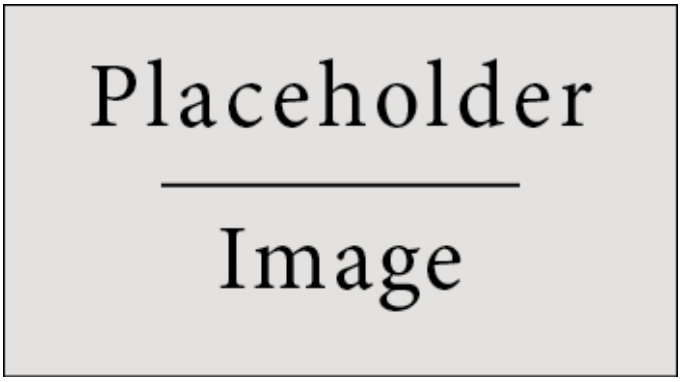

Figura 1: Figure caption.

(mensagem 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 abstração das *threads*).

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

4 Estudos de Caso

5 Considerações Finais

The most obvious source of experimental uncertainty is the limited precision of the balance. Other potential sources of experimental uncertainty are:

the reaction might not be complete; if not enough time was allowed for total oxidation, less than complete oxidation of the magnesium might have, in part, reacted with nitrogen in the air (incorrect reaction); the magnesium oxide might have absorbed water from the air, and thus weigh “too much."Because the result obtained is close to the accepted value it is possible that some of these experimental uncertainties have fortuitously cancelled one another.

Random citation Tyrväinen (2016) embeddeed in text.

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.

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

Tyrväinen, J. (2016). Concurrent solution for lamport clocks. `https://github.com/religiosa/lamportClocks`. Accessado em 14/06/2018.