Thread-based VS Event-driven servers

Alexandre Truppel (up201303442@fe.up.pt) Luís Abreu (up201206737@fe.up.pt)

***Abstract*—Exemplos de servidores TCP baseados em *threads* e em eventos são implementados em C (sem *non-blocking* IO). A implementação com base em eventos é feita com recurso a uma *queue* FIFO e *work threads*, e de duas maneiras: de raiz e usando a biblioteca GCD. As três implementações são testadas com várias configurações diferentes num Mac com um CPU *quad-core e 8GB de RAM.* A *performance* de cada implementação é medida e comparada. É concluído que, com as implementações efetuadas e no computador usado para os testes, *threads* pode ser até 10x pior que eventos.**

I. Introdução e Objetivos

Servidores TCP são absolutamente essenciais para o funcionamento correto e expedito da Internet. A sua implementação tem portanto um impacto muito grande e é de extrema importância. Várias arquitecturas são possíveis para a implementação de servidores concorrentes, sendo as principais baseadas em eventos ou baseadas em *threads*.

Os objetivos deste trabalho são:

- Desenvolver dois servidores TCP, um baseado em eventos e outro baseado em *threads.*
- Definir a sequência de operações que os servidores executam por cada conexão TCP. Essa sequência deve conter operações de IO e CPU.
- Executar os dois servidores, medindo e comparando a *performance* de ambos no espetro completo de carga.

Estes objetivos vagos foram evoluindo e tornando-se mais precisos ao longo do projeto. Os resultados dessa evolução serão descritos neste relatório.

II. Operações executadas pelo servidor

Em primeiro lugar foram decididas as operações que o servidor deve executar.

Para simplificar decidiu-se que cada cliente TCP, ao conectar-se ao servidor, faz um único pedido. Após a execução desse pedido pelo servidor, o cliente fecha a ligação. Sendo assim o número de pedidos ao servidor é igual ao número de clientes que se ligaram ao servidor.

Cada pedido é constituído por uma sequência de blocos de código que devem executar por ordem. Dividir o pedido em blocos de código bem definidos permite tanto uma implementação baseada em *threads* como em eventos, e permite também que ambas as implementações estejam a correr o mesmo código e possam portanto ser comparadas em termos de *performance*.

Uma das maiores razões para concorrência em servidores é o bloqueio que existe quando se acede ao disco. Sendo assim, uma boa implementação de um servidor deve, perante um pedido que ficou parado no acesso ao disco (e que portanto não está a utilizar o CPU), escalonar outro pedido para correr no CPU. Dessa maneira, qualquer comparação entre implementações deve ser feita com pedidos que contêm tanto blocos de IO como blocos de carga computacional no CPU.

Sendo assim, os blocos executados pelo servidor são os seguintes (por ordem de execução):

1. **Servidor recebe 8 Bytes enviados pelo cliente.**

Este valor é gerado aleatoriamente pelo cliente e será usado pelo servidor como *offset* de leitura e escrita num ficheiro de dados. Não há razão específica nenhuma para este valor ser gerado pelo cliente, nem para serem 8 Bytes. A razão principal deste bloco é tentar imitiar o uso real de um servidor TCP, em que se presume que haja um mínimo de comunicação entre o cliente e o servidor.

2. **Servidor acede a um ficheiro de dados no disco e copia parte do seu conteúdo para memória.**

Esta é a primeira operação de IO, neste caso de leitura: ler 80 KBytes de um ficheiro de dados (nos testes efetuados, com 1 MByte). Todos os pedidos lêem do mesmo ficheiro, mas o valor anterior de 8 Bytes aleatórios é usado como *offset*. Ou seja, cada pedido ao servidor lê secções diferentes do ficheiro. Isso é feito de modo a tentar evitar que o OS faça *caching* do ficheiro. Não há razão maior para serem 80 KBytes. Existe no entanto uma vantagem em não ser um número pequeno: para facilitar os testes, é vantajoso que cada bloco demore um mínimo de tempo a executar. Ler apenas 5 Bytes, por exemplo, seria demasiado rápido e não permitiria testar o servidor em regimes de carga elevados.

3. **Servidor calcula uma *hash* (1 Byte) dos 80 KBytes em memória.**

Esta *hash* é um simples *xor byte* a *byte* dos dados lidos. Tal como se implementou o *xor* podia ter-se usado qualquer outra operação nos dados, desde que fosse 100% carga no CPU, que é o objetivo deste bloco.

4. **Servidor escreve *hash* (1 Byte) no ficheiro de dados (mesmo *offset*).**

A segunda operação de IO, neste caso de escrita. A escrita é feita no mesmo ficheiro de dados, na posição do primeiro byte lido. Esta segunda operação de IO tem a vantagem de permitir tornar o servidor mais realista. É muito comum em servidores TCP reais haver pedidos a correr em paralelo que acedem a uma base de dados comum (por exemplo), onde alguns vão ler e outros vão escrever.

5. **Servidor envia *hash* para o cliente.**

A última operação é uma operação TCP em que o servidor envia ao cliente o resultado da *hash* (1 Byte). Isto assinala ao cliente que o pedido foi executado pelo servidor e o cliente fecha a ligação.

A sequência de operações executada pelo servidor é portanto resumida da seguinte forma: TCP ‣ IO-R ‣ CPU ‣ IO-W ‣ TCP. No entanto, é preciso chamar a atenção para um pormenor. Como será explicado à frente, são feitas medições da *performance* do servidor enquanto este corre. Essas medidas são guardadas num ficheiro de dados para análise posterior. O acesso e escrita nesse ficheiro é feito no fim da execução de cada pedido. Ou seja, o último bloco na sequência efetivamente contém TCP e IO-W. No entanto, numa execução "normal" (sem medidas de *performance*) deste servidor, o último bloco é de facto apenas TCP.

III. Implementação dos servidores TCP

A implentação de ambos os servidores foi feita em C, num só executável (para facilitar testes e evitar duplicação de código). O modo de funcionamento do servidor (*threads* ou eventos) pode ser escolhido com um argumento.

A. Thread-based

A implementação do servidor baseado em *threads* é simples. Existe um *main thread* que aceita as ligações TCP. Por cada cliente aceite o servidor cria um *thread* novo, e cada *thread* executa os 5 blocos de código em sequência. O *thread* desaparece

assim que o pedido é executado. Não são definidas prioridades entre *threads*, logo o seu escalonamento é deixado totalmente a cargo do OS.

B. *Event-driven*

A implementação baseada em eventos é mais complicada, como seria de esperar. Não foram usadas operações de IO assíncronas, por isso a implementação orientada a eventos continua a usar *threads* para ser concorrente. No entanto, ao contrário do servidor *thread-based*, esta usa um número fixo de *threads* e cada *thread* executa os pedidos bloco a bloco, em vez de executar um pedido completo de forma ininterrupta.

Existe um *main thread* que aceita os pedidos da ligação TCP e uma *workpool* com vários *work threads* (que é criada no início da execução do servidor). Por cada pedido aceite é criada uma máquina de estados para o executar. De seguida, o primeiro bloco de execução do pedido é enviado a um *dispatcher*. O *dispatcher* coloca o pedido numa *queue*. Sempre que um dos *work threads* está livre e a *queue* não está vazia, o *thread* retira o próximo bloco da *queue* e executa-o. O próprio bloco, no fim da sua execução, atualiza a máquina de estados. Na atualização, a máquina de estaudos envia para o *dispatcher* o próximo bloco a ser executado na sequência desse pedido. O ciclo repete-se até a máquina de estados indicar que o pedido foi concluído.

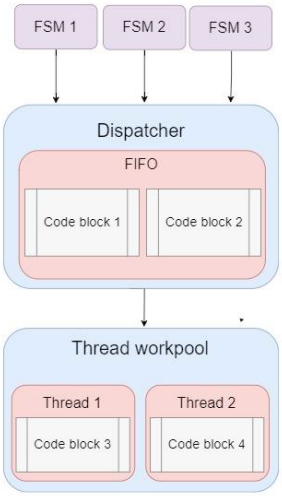


Fig. 1. Esquema da implementação de um servidor baseado em eventos com uma *workpool*.

1) *Implementação eventos-custom*

Foi desenvolvida uma implementação de raiz tal e qual como o explicado acima. O número de *work threads* pode ser variado. Foram implementadas duas *queues*: uma *queue* FIFO e uma *priority queue*, e ambas podem ser usadas apenas alterando o valor de um *#define* no código. No entanto, apenas foi testado o servidor com a *queue* FIFO. Usar uma *priority queue* significa desenhar um algoritmo para atribuir uma prioridade a cada bloco, e analisar esse algoritmo está fora do alcance deste trabalho.

2) *Implementação eventos-GCD*

A biblioteca GCD (*Grand Central Dispatch – libdispatch*), desenvolvida pela Apple, oferece muitas ferramentas para a execução otimizada de código em processadores *multi-core*. Uma das ferramentas principais é precisamente *multi-core serial/concurrent FIFO queues*.

Da Apple sobre GCD:

"GCD provides and manages FIFO queues to which your application can submit tasks in the form of block objects. Work submitted to dispatch queues are executed on a pool of threads fully managed by the system."

"A dispatch queue can be either serial, so that work items are executed one at a time, or it can be concurrent, so that work items are dequeued in order, but run all at once and can finish in any order."

Uma das ideias fundamentais da biblioteca é simplificar e aumentar a *performance* da implementação de uma *workpool* retirando o controlo dos *work threads* ao programador e dando-o ao sistema. A biblioteca também se adapta automaticamente consoante as características do *hardware* em que está a correr.

Visto que o GCD permite fazer exatamente o que é necessário para um servidor baseado em eventos (com *work threads*), mas é bastante otimizado, também foi usado para uma segunda implementação baseada em eventos. Assim é possível ter um termo de comparação para a implementação eventos-*custom* (que não usufrui da possível integração mais profunda que o GCD pode ter com o *hardware*).

IV. Medições feitas no servidor

De forma a quantificar a *performance* do servidor é necessário efetuar medidas enquanto este está a correr. Cada pedido no servidor tem uma execução com o seguinte formato (em que os tempos para cada bloco e entre cada bloco podem variar com a carga e com o modo de funcionamento):

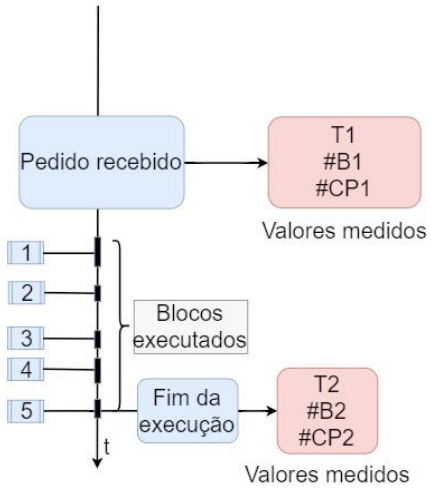


Fig. 2. Instantes em que são efetuadas as medições (por cada cliente).

A cada execução de um pedido estão associadas as medições de seis valores, como se pode ver na imagem. Três valores são medidos no momento em que o pedido é recebido pelo servidor; os outros três são medidos assim que o pedido é completado. As medições são as seguintes:

- **T1** e **T2**: *timestamps*
- **#B1** e **#B2**: número de blocos executados no total (de todos os pedidos) pelo servidor desde que foi ligado até esse momento
- **#CP1** e **#CP2**: número de clientes ligados ao servidor nesse momento

A partir dessas medidas é possível calcular, para cada cliente, o seguinte conjunto de valores:

- ***Time interval***: T = T2 – T1
- ***Block count***: #B = #B2 - #B1 = número de blocos executados no total (de todos os pedidos) pelo servidor nesse intervalo T
- ***Throughput***: TP = #B / T
- **Carga no servidor**: #CP = (#CP1 + #CP2) / 2 = número médio de clientes conectados ao servidor durante a execução do pedido

Estes valores são adicionados ao fim de um ficheiro de dados assim que são calculados (no fim do bloco #5).

V. TESTBENCH

A. Gerador de carga

Inicialmente seguiu-se o plano de gerar carga no servidor através de um segundo processo, escrito em C, que gerava ligações TCP para o servidor. No entanto verificou-se durante a fase de testes que usar TCP para criar pedidos no servidor causa vários problemas. Uma das principais dificuldades foi que a *stack* protocolar do OS, com um número grande de ligações TCP, começa a recusar executar mais conexões, o que impede o controlo do nível de carga no servidor para níveis de carga elevados. Outra observação é que usar TCP para testes causa demasiada variação nas medidas, sem dúvida devido a atrasos causados também pela *stack* protocolar. Como o objetivo é medir a *performance* das duas arquitecturas do servidor, é conveniente retirar das medidas influências externas como estas.

Sendo assim, para evitar TCP, foram feitas algumas adaptações à ideia inicial. O próprio executável do servidor, em vez de executar o *accept(...)* de ligações TCP (no *main thread*), executa o código gerador de pedidos. Esse código, em vez de criar uma ligação TCP para cada cliente (como seria o caso se fosse executado num programa separado, segundo o plano inicial), injeta um pedido no servidor. Os blocos de execução #1 e #5 (blocos TCP) são mudados: o bloco #1 passa a gerar os 8 Bytes aleatórios em vez de os receber por TCP e o bloco #5 passa a estar vazio (quase vazio, continua a conter o *overhead* do *logging* das medições). Dessa maneira, TCP é evitado mas tudo o resto mantém-se igual.

O código desenvolvido permite que o servidor corra em ambos os modos de teste (com ou sem TCP), mas só o modo sem TCP é que foi usado durante os testes.

B. Formas de carga

O gerador de carga do servidor consiste no seguinte código:

1. Gerar um novo cliente
2. Esperar X microsegundos
3. Repetir Y vezes

O valor de X é o que permite controlar a carga no servidor: valores altos de X resultam numa menor carga e valores baixos resultam numa maior carga. O valor de Y define quantos clientes são gerados (efetivamente define quantas medições são geradas e quanto tempo demora a correr o teste ao servidor).

Inicialmente foram executados testes com o valor de X constante ou aleatório entre um mínimo e um máximo. O problema que se descobriu com esses dois regimes foi que, dependendo dos valores de X, o servidor ou estava constantemente em regime de sobrecarga (ou seja, a receber mais pedidos do que consegue executar) ou sempre no contrário (ou seja, usando apenas uma fracção da capacidade total do computador). No entanto, uma das partes importantes da *performance* de um servidor (e que ajuda a preencher completamente o eixo horizontal nos gráficos de resultados) é durante as alturas em que o servidor transiciona entre sobrecarga e "sub carga" (em ambas as direcções).

Para obter resultados também nessas transições acabou por se usar uma variação sinusoidal de X, definida pelo seu valor máximo e mínimo. Assim é possível percorrer os vários regimes pois tem-se valores máximos e mínimos (mínimos e máximos de carga), e subidas e descidas (transições entre sobre e "sub" carga). Alguns dos testes também foram feitos com valores máximos e mínimos iguais, ou seja, X constante, para também obter resultados em verdadeiros regimes (não oscilatórios) de sobre e "sub" carga.

Os valores de limite da sinusóide de X podem ser passados ao executável do servidor por argumentos para simplificar a execução dos testes.

C. Testes efetuados

Todos os testes foram efetuados num único computador, um *MacBook Pro (15-inch, Late 2011)* com um processador *2,2 GHz Intel Core i7*, 8 GB de memória RAM e o macOS Sierra instalado. O processador em questão tem 4 *cores* com *hyperthreading*, para um total de 8 execuções "fisicamente concorrentes" de código.

O servidor foi executado no modo de teste sem TCP com 15 regimes de carga sinusoidais diferentes (pares mínimo-máximo diferentes) para cada um dos seguintes modos de funcionamento:

- *Threads*
- Eventos GCD
- Eventos *custom* com 4 *work threads*
- Eventos *custom* com 8 *work threads*
- Eventos *custom* com 16 *work threads*

No modo de funcionamento de eventos, cada regime de carga foi testado com Y = 2000 clientes. Já no servidor *thread-based* os testes foram mais difíceis. Devido à limitação do número máximo de *threads* no computador, foi apenas possível aplicar os 15 regimes de carga com 500 clientes cada. Valores de Y mais elevados bloqueiam o próprio OS.

VI. RESULTADOS & ANÁLISE

A. Gráficos obtidos

A partir do conjunto de dados obtidos para cada cliente (T, #B, TP e #CP) foram efetuados os seguintes gráficos:

- **T *versus* #B** – permite analisar se o tempo médio para executar cada bloco se mantém constante com o número de blocos
- **T *versus* #CP** – permite analisar como varia o tempo para executar cada pedido com a carga no servidor
- **TP versus #CP** – permite analisar a variação do *throughput* do servidor (em número de blocos executados) com a carga no servidor

Os gráficos resultantes podem ser consultados nos anexos (gráficos iniciais, pontos a laranja). Os tempos estão em segundos e a *throughput* em blocos por segundo.

B. Pós-processamento de dados

O objetivo de executar estes testes é comparar a *performance* dos vários modos de funcionamento, algo que se torna mais expedito quando os resultados estão todos sobrepostos no mesmo gráfico. No entanto, cada gráfico de cada modo pode conter até 30000 pontos, o que torna uma sobreposição eficaz impossível.

Sendo assim, foi necessário fazer um pós-processamento dos dados em cada um dos 15 gráficos iniciais. Em primeiro lugar foi feita uma média, de modo a associar a cada valor no eixo X apenas um valor no eixo Y. Depois foi feita uma *moving average* à curva resultante (para eliminar ruído e variações bruscas). Os resultados podem ser consultados nos anexos (gráficos iniciais, curva média a azul).

C. Gráficos finais

As curvas médias de cada um dos 15 gráficos foram então sobrepostas em três gráficos finais (T vs #B, T vs #CP e TP vs #CP). Esses gráficos podem ser consultados nos anexos (gráficos finais).

D. Análise

A partir dos gráficos obtidos é possível concluir um grande conjunto de informações relevantes:

- **A *performance* do servidor baseado em *threads* é 3x a 10x pior que qualquer implementação baseada em eventos.**

Isso é possível verificar em todos os gráficos. Este resultado pode ser devido a algum detalhe de implementação do servidor no

código C, mas o mais provável é que seja devido ao fenómeno de *thrashing*, que é pior quanto maior o número de *threads* usados.

- **A implementação *custom* (com 4, 8 ou 16 *threads*) é melhor que a que usa o GCD.**

Este é um resultado inesperado. Não nos é claro o porquê desta diferença. O conhecimento que temos da maneira como GCD opera alinha-se totalmente com a arquitectura da implementação *custom* (ambas usam também alocação dinâmica de memória). Uma das hipóteses é que a biblioteca não está otimizada para *queues* com 1000+ elementos, mas essa hipótese não é muito provável porque o gráfico T vs #B mostra que GCD é pior mesmo para valores de #B pequenos.

- **Nenhuma das nossas implementações baseadas em eventos consegue manter a *performance* em regime de sobrecarga.**

A teoria indica que um servidor baseado em eventos aumenta a *throughput* em proporção com o número de clientes até chegar ao máximo, momento a partir do qual se mantém constante e forma um patamar. No entanto, o gráfico TP vs #CP indica que as implementações baseadas em eventos deste trabalho não se comportam dessa forma. Todas elas (excepto *custom* 4 *threads*, por razões que se tornam aparentes na conclusão seguinte) têm uma subida até um máximo por volta dos 10 a 20 clientes, mas depois caem de volta para os seus valores iniciais. Este comportamento mais uma vez pode denunciar uma implementação incorreta, mas o mais provável é que se esteja a ver o resultado de *overhead* de memória

- **A escolha do número de *work threads* na implementação *custom* influencia a *performance.***

Segundo a teoria, uma implementação de eventos com 4 *threads* para um CPU com 8 "*threads* físicos" tem um máximo de *throughput* igual a metade do máximo de uma implementação com 8 ou mais *threads*. Por outro lado, uma implementação com mais do que 8 *threads* não atinge um máximo de *throughput* maior que o possível com 8 *threads*, mas tem um *overhead* maior (devido ao aumento do número de *threads*).

Apesar do valor não ser exatamente metade, esta mesma lógica pode ser verificada nos resultados obtidos. Nos gráficos TP vs #CP, *custom* 4 *threads* não tem quase pico de *throughput*, enquanto que os outros (8 e 16) têm. Mais ainda, apesar de 8 e 16 estarem muito próximos, 16 está claramente abaixo de 8, devido ao *overhead* extra. Nos outros gráficos (T vs #B e T vs #CP) é possível também verificar que 8 é o melhor, depois 16 e depois 4, com excepções muito pontuais.

- **O tempo (médio) por bloco exibe várias retas horizontais para servidores baseados em eventos.**

Nos vários gráficos T vs #B é possível ver, para valores de #B altos, algumas retas horizontais. Este fenómeno foi difícil de perceber e para chegar a uma conclusão foi necessário analisar os resultados cliente a cliente nos testes efetuados. Uma explicação resumida é dada de seguida.

Os maiores regimes de carga nos servidores são gerados colocando o valor de X (tempo de espera entre pedidos novos) muito pequeno. Em alguns casos, tão pequeno que todos os 2000 clientes são criados quase ao mesmo tempo. Isto significa que, dado o funcionamento do *dispatcher*, o bloco #1 de todos os 2000 clientes é adicionado à *queue* FIFO antes sequer do primeiro bloco no FIFO acabar de executar. Quando esse primeiro bloco acaba, adiciona ao fim do FIFO o bloco #2 para o seu pedido. O ciclo repete-se, criando um padrão na *queue*: à cabeça estão os 2000 blocos #1, depois aparecem os 2000 blocos #2, depois todos os #3, etc. Ou seja, quando os clientes aparceram todos de uma vez, os vários tipos de blocos (#1, #2, #3, #4 e #5 na sequência de cada pedido) não ficam misturados, mas sim todos juntos e por ordem.

Este funcionamento tem várias consequências:

- Alguns blocos demoram menos tempo a executar que outros. Sendo assim, é possível ter um cliente que no tempo T tenha #B = 8000 e outro que no mesmo tempo T tenha #B = 10000. Para isso basta apenas que no segundo caso se tenha executado mais blocos rápidos que no primeiro. Isso explica as linhas horizontais nos gráficos.
- Um servidor baseado em eventos com a arquitectura implementada (uma *queue* FIFO e *work threads*) nunca será o mais eficiente possível em regimes de grande carga. Por exemplo, se alguns *work threads* bloquearem em IO, executar um bloco de CPU para manter o CPU ocupado pode não ser possível. Isto acontece, não porque não há blocos de CPU na *queue* que possam correr nesse momento, mas porque esses blocos estão atrás de mais blocos de IO (que serão os próximos a sair do FIFO porque os blocos não estão misturados de forma equilibrada).

Este fenómeno chama a atenção para um problema na arquitectura escolhida para o servidor baseado em eventos. Esse problema pode ser resolvido mudando a arquitectura.

Uma solução possível é, em vez de ter um FIFO concorrente global e uma *workpool*, ter um FIFO série por cliente e todos esses FIFOs usarem a mesma *workpool* de forma concorrente. Assim minimiza-se o atraso que um número elevado de clientes causa em cada cliente. Desta maneira também se torna mais difícil ter parte do servidor bloqueado em IO sem ser possível ir buscar operações CPU às *queues* para executar.

Outra solução é ter na mesma uma única *queue*, mas essa *queue* deixar de ser FIFO e passar a ordenar os blocos por prioridades. Um bom algoritmo de escolha das prioridades talvez consiga resolver estes problemas.

VII. Conclusões

Consideramos que todo o trabalho foi executado e concluído com sucesso. As implementações dos servidores foram além do que era esperado (não só se implementou completamente um servidor baseado em eventos, como se fez isso de duas maneiras diferentes). Os testes executados foram exaustivos (várias configurações dos servidores) e permitiram retirar resultados interessantes.

A. Críticas e possível trabalho futuro

Existem váris áreas onde o trabalho pode ser expandido e melhorado:

- **Implementação:**
  - Testar outras arquitecturas de servidores baseados em eventos, tal como descrito na secção anterior.
  - Investigar como implementar servidores baseados em eventos que mantém o *throughput* mesmo em regimes de carga elevada
  - Melhorar a implementação baseada em *threads* de modo a produzir melhores resultados (ou pelo menos verificar se é possível fazê-lo).
  - Investigar mais profundamente a razão pela qual a biblioteca GCD é mais lenta.
- **Testes:**
  - Executar testes noutros computadores, com outros sistemas operativos e *hardware* e verificar se as consluões se mantém.

B. Contribuição de cada elemento

- **Alexandre (60%):**
  - Implementação dos servidores TCP (3 modos de funcionamento e código de apoio – *threads, mutexes, queues, workpool, sockets* TCP, etc)

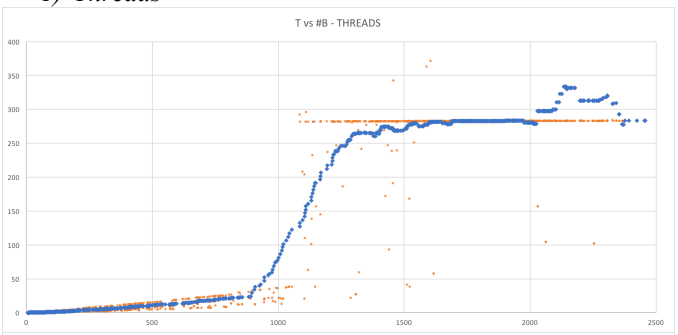
- Execução dos testes ao servidor no computador
- Escrita da apresentação e relatório

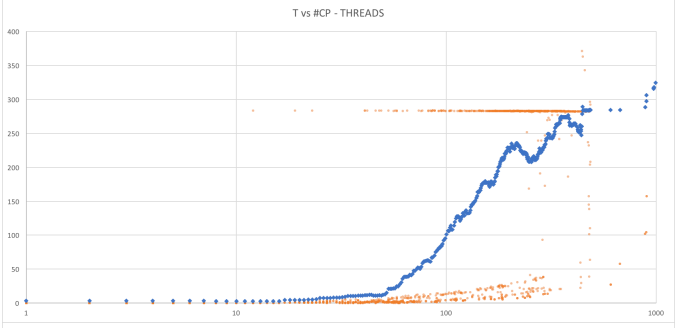
- **Luís (40%):**
  - Compilação e análise dos resultados
  - Escrita da apresentação e relatório

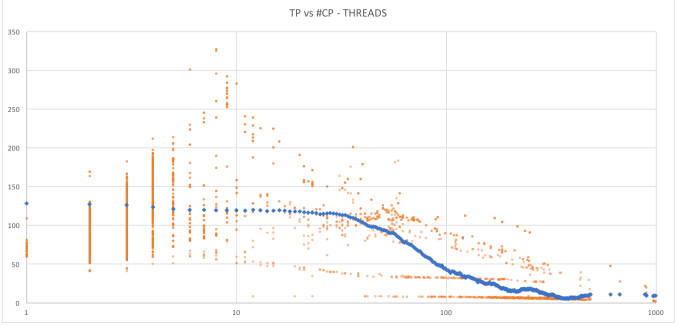
VIII. Anexos

A. *Gráficos iniciais (pontos laranja) com curva média (azul)*

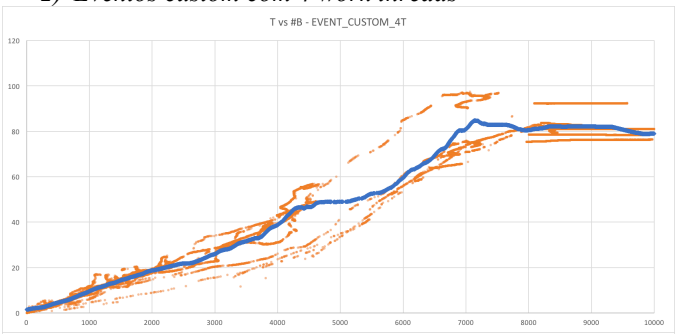
1) *Threads*

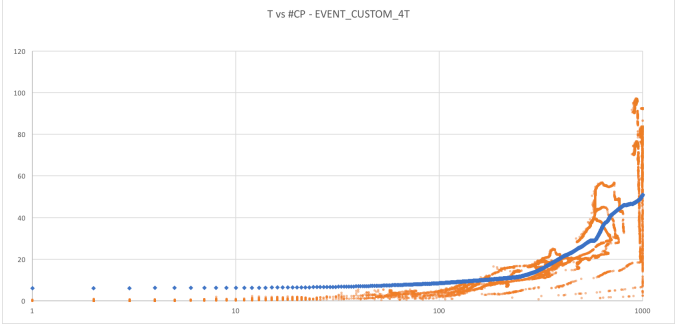


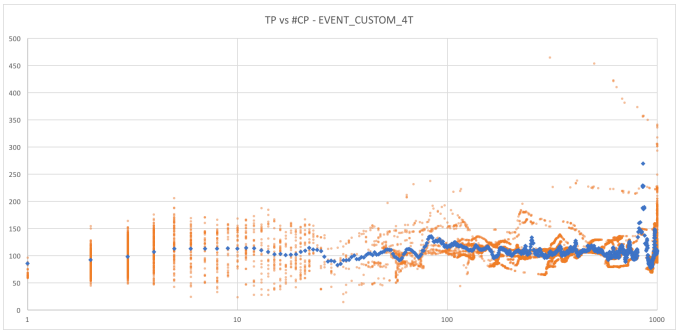




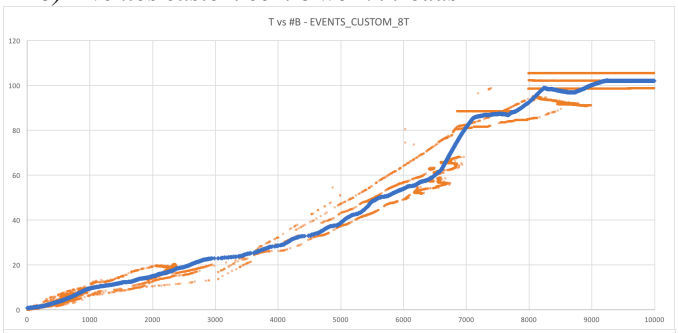
2) *Eventos custom com 4 work threads*

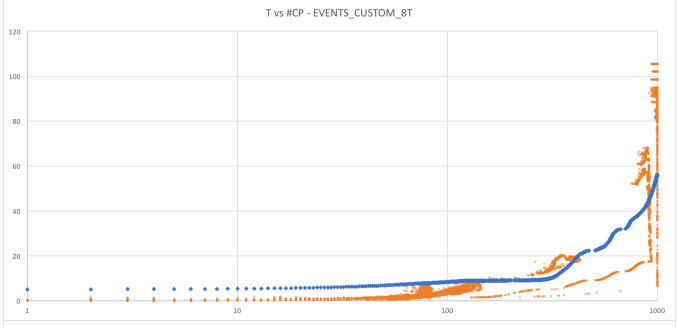


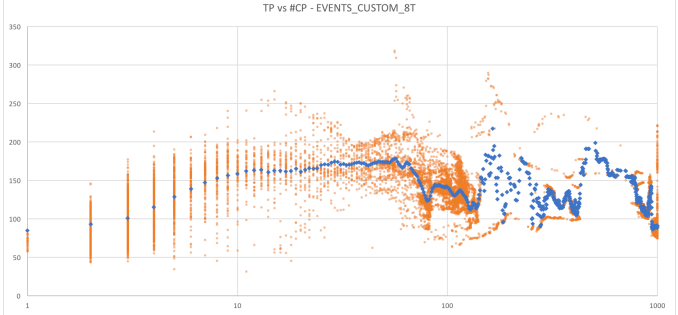




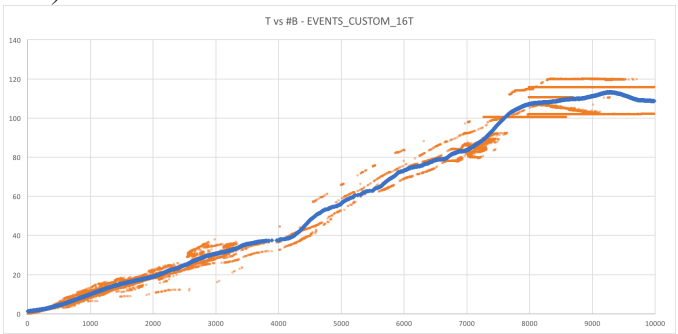
3) *Eventos custom com 8 work threads*

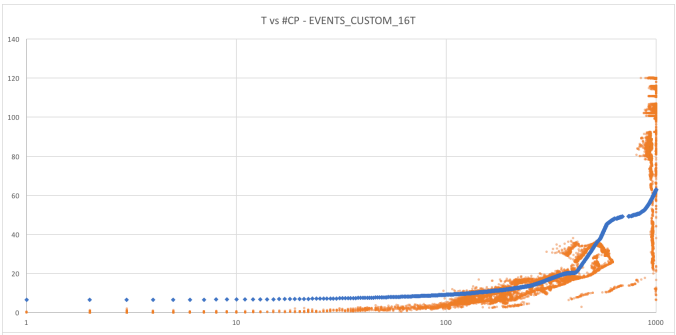


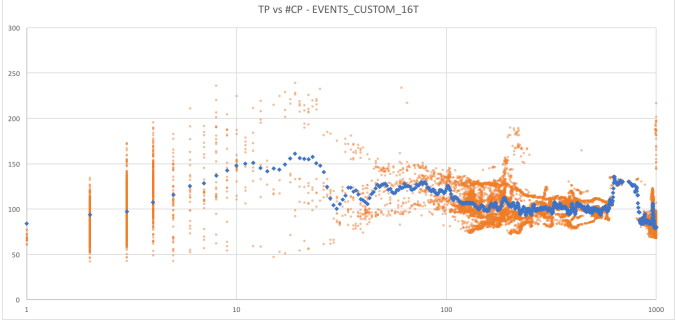




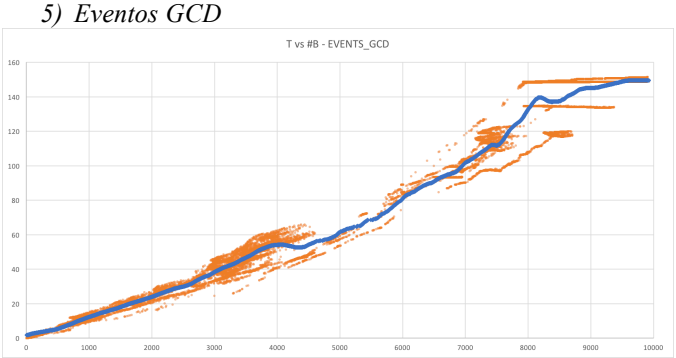
4) *Eventos custom com 16 work threads*

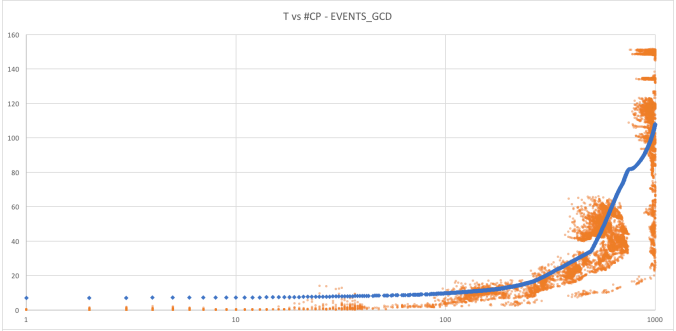


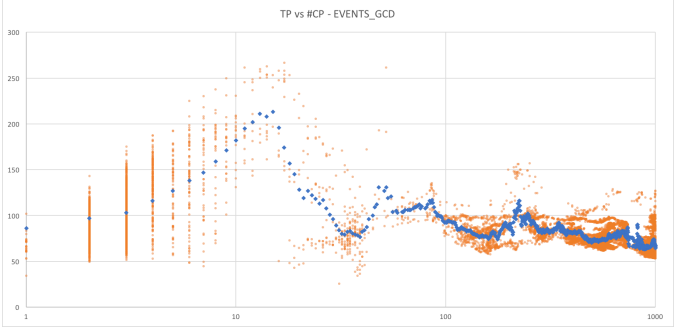




5) Eventos GCD

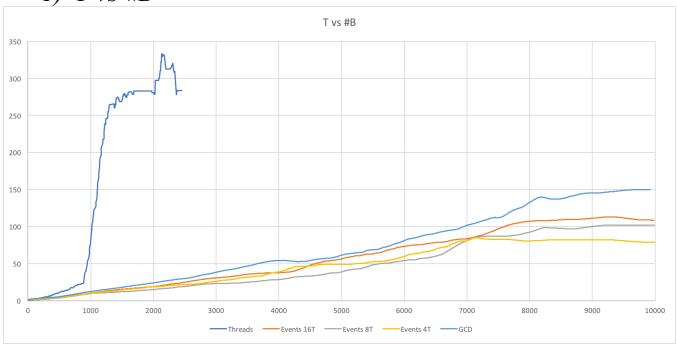




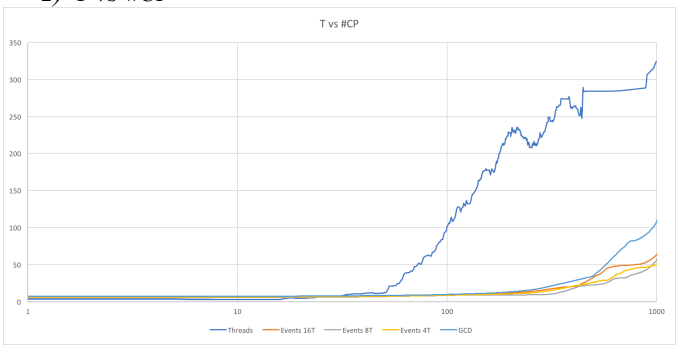


B. Gráficos finais

1) T vs #B



2) T vs #CP



3) TP vs #CP