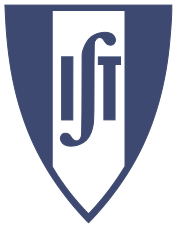


INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

PSis 2011/2012

MEEC - MEAer

Prof. Luís Silveira

Yet **A**nother **S**imple **C**alculator

14 de Maio de 2012

64956 Bruno Santos *bruno.dos.santos@ist.utl.pt*
74570 Flávio Schuindt *flavio.silva@ist.utl.pt*

Conteúdo

1 Introdução

Este relatório destina-se a cobrir a implementação em *C* de uma calculadora em rede numa estrutura de cliente - servidor, implementada segundo as normas ANSI C99 e POSIX.

Na nossa abordagem, optámos por uma implementação monolítica de *IO* síncrono para o cliente, e uma estrutura altamente paralelizada e completamente assíncrona para o servidor. Estas opções serão discutidas ao longo deste relatório, assim como algumas funcionalidades previstas, mas que não chegámos a implementar.

A estrutura deste documento corresponderá a uma descrição tanto do cliente como do servidor, seguidos de uma análise crítica ao seu desempenho na forma final, submetida juntamente com este relatório.

Neste relatório, incluiem-se ainda em anexo um diagrama do conceito abordado (A), assim como os manuais tanto do cliente (B) como do servidor (C). Contudo uma explicação sumária do seu funcionamente é dada também no corpo do documento.

O desenvolvimento deste projecto foi constantemente registado num repositório *git* online que pode ser consultado em *bitbucket.org*.

2 Cliente

2.1 Chamada

A invocação do cliente segue os requisitos do enunciado, bastando indicar o endereço do servidor e a porta de escuta. A falha em passar dois argumentos corresponde a uma falha directa, mas a correcção destes parâmetros não é feita senão mais tarde quando e se o utilizador decidir iniciar sessão.

```
yascC.exe <hostname> <port>
```

Opcionalmente o utilizador poderá ainda fornecer um ficheiro de texto com comandos a executar cuja sintaxe deverá ser idêntica à usada na linha de comandos. Para esse efeito deverá ser dado o directório do programa como parâmetro extra, precedido de `-f`. A verificação da extensão de ficheiro de texto não é verificada, pelo que o ficheiro usado não tem de ser necessariamente `.txt`.

```
-f <filename.txt>
```

Ainda no processo de chamada o utilizador pode configurar duas *flags* adicionais: `-l` e `-g`.

A primeira destina-se a suprimir o *output* da linha de comandos e registá-lo na totalidade num ficheiro `log.txt`, excepto erros críticos que continuam a ser dirigidos para o terminal. Esta *flag* não pode ser alterada em qualquer outro ponto da execução do cliente e destina-se apenas a permitir uma melhor análise do *output* do programa, sobretudo em situações em que um ficheiro de comandos é usado.

A segunda *flag* permite apenas que o modo *Debug* seja iniciado no arranque, mas o utilizador pode em qualquer momento alternar entre os dois modos de *output*.

Com a excepção do endereço do servidor e da porta de escuta, nenhum destes parâmetros é obrigatório nem tem ordem fixa.

A execução do cliente num directório que não contenha `/doc` não permite o acesso ao ficheiro de ajuda durante a execução.

2.2 Interface

A interface com o cliente é feita à custa dos comandos:

- ◇ Controlo de sessão:
  - · I
  - · K
- ◇ Operações em *stack*:
  - · +, −, ∗, /, %
- ◇ Controlo de *stack*:
  - · P
  - · T
  - · R
- ◇ Outros:
  - · G
  - · HELP
  - · EXIT

Os comandos `I` e `K` permitem iniciar e terminar, respectivamente, a ligação ao servidor. Esta operação pode ser feita em qualquer ponto ponto da execução do cliente, e depois de terminada a sessão o cliente pode voltar a iniciá-la sob a penalidade de ficar bloqueado na fila do `listen()`. No caso de uma sessão ter sido previamente iniciada, `I` deverá limpar o *stack* com retorno positivo `V0`.

As operações aritméticas, todas elas efectuadas sobre números inteiros de 32 bits, retornam sempre `V0` excepto em caso de overflow, underflow ou divisão por 0.

Para visualizar e controlar o *stack*, tem-se ainda os comandos `P`, `T` e `R`. Estes permitem saber o tamanho do *stack*, resultados parciais ou números acabados de introduzir e resultados totais, respectivamente. O modo mais simples de reinicializá-lo é, como foi dito, com o comando `I`.

A introdução de valores no *stack* é feita simplesmente introduzindo números inteiros em base decimal ou, em alternativa, em base octal na forma *0###* ou na base hexadecimal na forma *0x###*.

Por fim, os restantes comandos destinam-se a ligar e desligar o modo *Debug*, a abrir um ficheiro na própria linha de comandos, e a fechar o cliente. Para a leitura do ficheiro de ajuda, tem ainda de ser tido em conta o directório de execução, conforme explicado na secção 2.1, e vale referir que `EXIT` chama internamente `K` por forma a avisar o servidor de que fechou sessão.

O *output* gerado é sempre precedido de sinalética própria do tipo de mensagem em questão:

◇ `>>` Erro que quebra funcionalidade do cliente.

◇ `: :` *Output* normal que inclui algumas mensagens de erro.
◇ `DEBUG:` *Output* extra do modo *Debug*.

Num erro crítico o cliente aborta com a mensagem de erro, mas em qualquer outro caso o cliente deverá apenas informar do erro e continuar a executar.

Para maior detalhe relativo à interface com o utilizador, pode ser consultado o manual do cliente.

2.3 Estrutura lógica

O cliente é constituído por um único fio de execução durante todo o seu tempo de execução. Permanece num ciclo quási infinito conforme é mostrado no trecho de código seguinte.

De notar que no código abaixo, `fin` identifica a linha de comandos ou o ficheiro de comandos passado ao programa.

```
while( fgets(line,MAX_LINE,fin) != NULL ) {
    parse_line(line);
}
```

Neste ciclo, que resume bem o programa, o cliente limita-se a analisar o *input* e a tratá-lo de forma serializada por ordem de introdução. Tal inclui a apresentação de mensagens relevantes que obtenha do servidor.

Este tratamento em série do *input* leva a que numa situação de bloqueio à espera do servidor se reflicta num cliente inutilizado.

2.4 Meios de comunicação

O cliente faz uso de um único socket para comunicação com outro processo, potencialmente remoto: o servidor. Usa o protocolo TCP/IPv4 por facilidade de teste na máquina de desenvolvimento, e todas as operações executadas sobre esse socket são síncronas.

Esta abordagem facilita a interface com o utilizador, sendo mais fácil gerir qual a resposta de cada comando individual. Uma abordagem assíncrona exigiria um buffer de mensagens trocadas entre cliente e utilizador e um maior esforço na sua apresentação.

Por outro lado, a notação polonesa inversa sobressai sobretudo pela sua fácil interpretação síncrona, pelo que esta implementação adequa-se bem aos requisitos do projecto.

2.5 Desenvolvimento futuro

O cliente neste porjecto é uma entidade relativamente simples, por definição, mas um conjunto de melhorias podia ser posto em prática para omelhorar.

A melhoria mais óbvia seria implementar assincronismo no envio de mensagens e tratamento de respostas, o que levaria a um problema bastante mais complicado tanto do lado do cliente, como mesmo do servidor.

Este assincronismo nunca esteve nos planos para este projecto, mas era um ponto de relevo se quiséssemos expandir o leque de possibilidades para a interface com o servidor.

3 Servidor

3.1 Chamada

A invocação do servidor segue estritamente os requisitos fornecidos pelo enunciado do projecto, bastando indicar como argumento a porta de escuta na qual o servidor deverá esperar clientes.

```
yascS.exe <port>
```

O utilizador pode ainda escolher se deseja que o modo verboso seja activado no servidor passando a *flag* `-v` na invocação do mesmo. O modo verboso permite que seja mostrado na saída padrão do servidor informações relativas à entrada e saída de clientes e ao lançamento de novos *workers* no *pool* de *threads*. A activação ou desactivação do modo verboso pode ser feita a qualquer momento pelo administrador do servidor, não sendo vinculativo o uso da *flag*.

A execução do servidor num directório que não contenha `/doc` não permite o acesso ao ficheiro de ajuda durante a execução.

3.2 Interface

O servidor disponibiliza uma interface para o administrador do mesmo e esta é feita à custa de comandos:

- ◇ Gestão do servidor:
  - · M
- ◇ Outros:
  - · V
  - · HELP
  - · F

O comando `M` é o único comando de gestão disponível. Este permite "tirar uma foto"do instante actual do servidor, isto é, quais clientes estão conectados e como está a pilha de cada cliente individualmente. O resultado é exibido na seguinte forma:

```
IP                [stack]
xxx.xxx.xxx.xxx   [stack top, ..., stack bottom]
```

Em que o campo IP é um endereço IPv4, e os valores de *stack* são inteiros de 32 bits com sinal compreendidos entre `INT_MIN` e `INT_MAX`, formatados em base decimal.

De notar que o comando `M` é um comando dispendioso em termos de performance, como será visto na secção 3.3. O seu uso constante bloqueia o tratamento dos clientes.

O comando V permite ligar ou desligar o modo verboso. Correntemente, este modo permite ter melhor noção do que está a acontecer no servidor em tempo real, mas só no que diz respeito à entrada e saída de clientes, e ao lançamento de *threads*.

Os comandos restantes destinam-se a abrir o manual do servidor e a fechar o servidor. Tal como no caso do cliente, é necessário que o manual esteja ao alcance do programa conforme discutido na secção 3.1.

Além das formas de *output* já abordadas, o servidor no ecrã resultados de erros internos, que invariavelmente levam à terminação do programa.

3.3 Estrutura lógica

O esqueleto do servidor é constituído por três unidades computacionais distintas que controlam todos os serviços prestados. São elas o servidor Master, o *Manager* e o *pool Manager*.

Vale referir que não lançamos novos processos, por serem mais pesados e não oferecerem nenhuma vantagem em relação às *threads* no contexto do nosso projecto. Isto é válido desde que, claro está, os problemas de sincronização sejam tidos em conta.

Threads, por outro lado, lançamos muitas além das três referidas. O *pool Manager* encarrega-se de criar ou matar *threads* com o único propósito de atender aos clientes – *threads* chamadas de *workers* ou *slaves* a partir deste ponto.

A razão por detrás desta abordagem foi um conceito de design que definimos bem cedo: queríamos uma *pool* de *threads* em que cada *worker* fosse capaz de atender a vários clientes.

Como os clientes nem sempre estão a executar comandos, não só não faz sentido ter unidades computacionais bloqueadas no `read()`, como não precisamos de tantas *threads*. Daqui vem uma grande vantagem na nossa implementação que é o facto de que podermos ter muitos mais clientes do que *threads*.

Aqui discutimos várias abordagens, uma delas um sistema em que o cliente estabelecia uma conexão por cada mensagem que enviava. Nesta implementação, para controlar a identidade dos clientes (possivelmente residentes na mesma máquina) seria necessário um sistema adicional de *tickets* para os associar a um dado stack. Aqui não só o servidor executava num nível de abstração do cliente maior, como as workers estariam a lidar directamente com uma lista de pedidos e não de clientes, com simplificações óbvias no esquema de atendimento que viémos a implementar. Esta solução, enquanto que muito versátil e poderosa, tinha as suas desvantagens.

Em primeiro lugar, o tempo de desenvolvimento seria relativamente grande, pois envolvia uma gestão de stacks muito mais complexa. Tal complexidade viria sobretudo da poluição do servidor com dados de clientes que saíssem sem fechar sessão explicitamente.

Em segundo lugar, para o tamanho das mensagens trnasmitidas, calculámos, sem fazer testes contudo, que o *overhead* introduzido pelo abrir e fechar do socket seria comparável ao tempo útil da ligação em si. O efeito sendo a perda generalizada de *performance*.

E por último, o esquema de aceitação de ligações estaria exposto a um possível impasse: numa situação de muitos clientes, um cliente com sessão iniciada não conseguiria passar pelo `accept()` e estaria a concorrer directamente com clientes

sem sessão iniciada, sendo a única diferença a posse de um *ticket*. Ora, o servidor teria de receber o cliente, e caso a lotação do servidor estivesse esgotada, comparar o *ticket* com uma lista e aceitar ou rejeitar o cliente com uma mensagem apropriada. Aqui não só os clientes com *ticket* teriam dificuldade em executar, como o servidor estaria activamente a rejeitar novos clientes, gastando recursos para tal.

Posto isto, optámos por uma solução mais modesta em que mantemos uma lista de clientes activos, e um socket dedicado a cada um desses clientes. Contudo esta implementação sacrifica alguma simplicidade na sincronização dos vários fios de execução e, sobretudo, a conveniência de bloquear automaticamente na chamada ao `read()`.

Definindo as funções de cada fio de execução, o *Manager* tem a única função de atender aos comandos do administrador, partilhando algum do código da função de *parsing* do cliente. Executa maioritariamente de forma completamente paralela, excepto quando o administrador pede a listagem de clientes com sessão iniciada. Aí, as condições de corrida são importantes e levou-nos ao uso de um *mutex* para controlar o acesso a esses recursos.

Analisando o código, a região crítica na função `print_client_info()` é na verdade um pouco inferior à região protegida com a tranca. Tal deve-se a uma tentativa de minimizar as hipóteses de uma outra *thread* interromper a listagem, permitindo uma melhor visualização da lista. Dito isto, o comando `M` destina-se a efeitos de debugging e administração, pelo que o seu peso deverá ser evitado em condições de funcionamento normal.

O módulo Master fica bloqueado grande parte do seu tempo de execução posterior ao setup do servidor. Ao aceitar uma nova conexão de um cliente, o Master server invoca a função `add_client()`, colocando-o na lista de clientes com sessão aberta. É importante ressaltar que o número máximo de clientes a serem aceites está limitado estaticamente por `MAX_CLIENTS`. Se o número de clientes na lista atingir esse máximo, então o servidor mestre pára de aceitar novas conexões e fica em *idle*, dormindo por um tempo definido em `DOORMAN_DOZE`.

Em vez de bloquear de tempo a tempo, uma outra solução para o Master seria controlar com primitivas de sincronização o número de clientes activos e sinalizar o Master quando este valor descesse novamente abaixo do valor máximo estipulado. Contudo, esta é uma situação de corrida que não nos convém corrigir. O facto de que o Master pode rejeitar uns poucos clientes por alguns segundos porque se baseou numa contagem desactualizada é irrelevante num cenário de várias centenas senão milhares de utilizadores. Assim o enquanto que o servidor ainda executa por vezes apenas para se bloquear novamente acaba por ser uma boa situação em relação a bloquear todas as *threads* para ler um inteiro, e adicionar testes ao número de

clientes dentro das *workers* pois seriam estas quem teriam a capacidade de gerar o sinal.

Por fim, o *pool Manager* tem a função de capataz responsável por gerir o esforço computacional associado ao atendimento de um dado número de clientes. Este dá origem a uma *pool* de *threads* que mantém de forma dinâmica. O número exacto de *threads* que são mantidas em execução é actualizado segundo uma taxa definida por `pool_REFRESH_RATE` numa função directa do conjunto de valores:

- ◇ `MIN_WORKERS`
- ◇ `MAX_WORKERS`
- ◇ `MAX_CLIENTS`
- ◇ `pool_HYSTERESIS`
- ◇ `CLIENTS_PER_SLAVE /* divisão inteira */`

Toda a aritmética envolvida no controlo da *pool* é feita sobre inteiros, com resultados inteiros, oferecendo portanto alguma sensibilidade aos valores atribuídos a cada uma das constantes. Esta opção é rápida e eficaz, mas obriga a algumas considerações. Por análise cuidada do código é possível perceber que situações indesejáveis podem acontecer.

Nomeadamente, o número de *slaves* nunca pode deve ser inferior a um, sob a penalidade de matar uma *thread* para lançar outra de seguida. Esta situação verifica-se por causa das trancas necessárias à correcta execução das *threads* que impedem que estas sejam interrompidas em dados pontos da execução em benefício do cliente.

Outro exemplo é para uma relação inteira par de clientes por *thread* (e.g. dois clientes por *thread*) com uma histerese nula. Nesta situação o *Manager*, por evitar demasiados testes, vai entrar num modo de ressonância quando tem metade dos clientes máximos, lançando uma *thread* e matando-a de seguida.

Os valores sugeridos para as constantes que condicionam a *pool* foram atingidos por benchmarking, ainda que limitado a um computador único. Certamente poderão ser ajustados para uma dada carga de trabalho por forma a optimizar os recursos da máquina que hospeda o servidor sendo portanto uma *pool* bastante configurável.

Resumindo o atendimento a um cliente, inicialmente, o gerenciador do *pool* lança um número mínimo de *workers*, que poderia até ser nulo, com as implicações já discutidas. A partir daí, cada *worker* fica num loop infinito varrendo sequencialmente uma lista de clientes. O facto de ser sequencialmente evita simplesmente que um cliente seja deixado por atender, mas não sendo aqui tão eficiente como esquemas em que uma *thread* fica bloqueada num cliente. Isto porque um cliente só por

existir vai atrasar o processamento dos outros, mesmo quando não envia nenhum comando.

Para percorrer os vários clientes, as *threads* têm de se sincronizar por uma lista global, obrigando a um acesso em exclusividade, utilizando duas estrutura de dados. Uma delas correspondendo à lista em si, e outra a um descritor que permite acesso directo a pontos chave da lista sem ter de a percorrer por completo. A vantagem deste descritor surge para listas muito grandes, em que é mais fácil manter a informação contida no descritor do que percorrer a lista toda de cada vez. Essa outra opção pesaria ainda mais por se tratar de uma estrutura partilhada de acesso em modo de exclusão mútua.

As esturturas que permitem a identificação e atendimento dos clientes são:

```
typedef struct clients_descriptor {
    struct client *first;
    struct client *last;
    struct client *next;
    int count;
} CLIENTS_DESCRIPTOR;

typedef struct client {
    int fd;
    char IP[INET_ADDRSTRLEN];
    struct client *next;
    struct client *previous;
    struct stack_descriptor *stack_desc;
} CLIENT;
```

De notar que o mesmo conceito de descritor é usado no *stack*, mais uma vez com prováveis perdas de performance para stacks pequenos, mas que são ao mesmo tempo mais invariáveis ao seu tamanho.

Depois de escolhido um cliente da lista, as *threads* são completamente iondependentes entre si e livres de qualquer problema de sincronização. Um caso interessante surge quando, havendo mais *workers* do que clientes, há mais do que uma *thread* a processar o mesmo cliente. Nesta situação só uma é que completa com êxito, não surgindo problema, ainda assim, o servidor foi desenhado tendo em mente vários clientes por cada *worker*.

Ultrapassado o problema de sincronismo, um dos desafios de ter uma só *thread* capaz de dar resposta a todos os clientes ao mesmo tempo, impica o uso de funções de *IO* assíncronas. Em particular precisamos agora de desbloquear as *threads* da função `read()` e conforme tenham lido alguma coisa ou não do socket, mandá-las avançar para outro cliente na lista. Este problema tem uma solução trivial, fazendo uso da função `fcntl()`, mas que levanta outro problema: ver secção 3.5.

3.4 Meios de comunicação

O servidor, por ter um esquema de múltiplos fios de execução, tem obrigatoriamente esquemas de comunicação internos a acrescer aos *sockets*.

Internamente, para a passagem dos *file descriptors* aos *workers*, equacionou-se um mecanismo com um *pipe* comum a todas as *threads*, mas o esquema que pretendíamos exigia a presença de certas estruturas para listar os clientes, como já foi referido. Por essa razão, o principal meio de comunicação entre os vários fios de execução são na verdade variáveis globais de acesso em regime de exclusão mútua, conseguida através de *mutexes*, incluindo um *mutex* condicional para parar as *threads* quando não há clientes em espera.

Para um outro fim, o *pool Manager* necessita de comunicar com os *workers* que lança. Isto serve o intuito de lhes indicar que devem morrer.

Um sinal é aqui a melhor opção, no entanto, não queremos que um sinal interrompa o tratamento de um cliente. Por isso bloqueamos o sinal dentro das *threads* mal estas são lançadas permitindo um tratamento assíncrono. No fim do tratamento de um cliente verificamos se foi lançado o sinal apropriado, `SIG_INT` neste caso, e matamos a *thread* que, tendo sido configurada como *detached* não depende de nennhum outro fio de execução para sair e libertar os seus recursos computacionais.

3.5 Desenvolvimento futuro

O principal elemento em falta no servidor na sua implementação actual é a espera activa sobre a execução dos clientes.

Devido a tornarmos as operações de *IO* sobre os *sockets* assíncronas, ficamos com uma *pool* de *threads* que nunca pára de executar, num loop de espera activa de que um cliente escreve uma mensagem num socket.

Paradoxalmente, o nosso servidor não consome mais CPU com maior carga de trabalho – apesar de o fazer com maior número de clientes. Isto é na verdade mau sinal, porque na verdade ele consome sempre o máximo (perto do máximo pelo menos) dos recursos que lhe são disponibilizados. A excepção à regra verifica-se quando não há nenhum cliente na fila, caso em que um *mutex* condicional bloqueia as *threads*, independentemente do seu número.

A forma de o evitar e contornar esta falha grave poderia passar por duas estratégias, não mutuamente exclusivas.

A primeira, com recurso a funções da família do `select()`. Nesta abordagem as *threads* ficam bloqueadas e, na chegada de uma comunicação, poderiam ser lançadas a percorrer a lista de clientes uma única vez. Uma ou mais *thread* iriam completar com sucesso o atendimento dos clientes que tinham comunicado alguma coisa, e todas as outras falhavam, mas, invariavelmente, todas se bloqueavam novamente. Esta é a solução mais plausível a curto prazo.

Uma segunda alternativa mais apelativa, mas que podia e devia ser combinada com a anterior, permite ainda maiores ganhos de performance em períodos de pouca actividade. Ao invés de controlar o número de *threads* na *pool* estaticamente, poderíamos criar uma variável que dava conta da entropia do sistema. Quantas mais *threads* terminassem sem atender a nenhum cliente, mais *threads* eram mortas libertando os recursos associados, e vice versa.

Neste esquema, a *pool* não só seria altamente configurável, mas teria um elemento de auto aprendizagem em que o único factor de ajuste seria um ganho a definir pelo administrador.

Uma forma possível de implementar esta segunda alternativa era incrementar uma variável afectada de um ganho positivo maior do que um, para evitar operações dispendiosas em vírgula flutuante, e decrementar com um ganho potencialmente distinto. A incrementação seria executada, por exemplo, no atendimento bem sucedido a um cliente, e decrementada na detecção de um cliente parado.

A partir daqui, o *pool Manager* teria apenas de matar ou criar *threads* para as manter a oscilar com este valor. Para evitar efeitos negativos com demasiadas criações de *threads* ou eliminações, bastaria manter o esquema de histerese actual.

Em qualquer um dos casos a implementação do sinal que manda terminar uma *thread* teria de ser revista, pois se as *threads* não estivessem em execução, não poderíamos esperar que elas terminassem logo no esquema actual.

4 Desempenho

Conscientes de que o nosso projecto está longe de ser eficiente no uso de recursos, pelos problemas já descritos, não deixamos de esperar um bom desempenho no atendimento aos clientes. Aliás, o ponto onde o desempenho é sub-óptimo, é precisamente quando o servidor tem clientes registados, mas nenhum requisita qualquer serviço.

Com isto em mente desenhámos um ficheiro de comandos suficientemente longo para que a sua execução não fosse demasiado rápida, e bombardeámos o servidor com clientes a ler os comandos directamente desse ficheiro.

Os parâmetros de ajuste da *pool*, completamente arbitrários, foram:

```
#define MIN_WORKERS 1
#define MAX_WORKERS 500
#define MAX_CLIENTS 1000
#define CLIENTS_PER_SLAVE ( MAX_CLIENTS / MAX_WORKERS )
#define POOL_REFRESH_RATE 1     /* [seconds] */
#define POOL_HYSTERESIS 2
```

No directório `/Init` é apresentado o output do comando `M` no servidor a meio da execução. Não é apresentado aqui pela sua extenção. Ao todo lançámos por volta de 200 clientes, e todos eles terminaram a execução em sucesso.

Testes com `MAX_CLIENTS` inferiores ao número de clientes lançados, proporcionaram resultados igualmente positivos.

De notar que estes testes foram executados, por simplicidade, numa única máquina de teste, a correr simultâneamente tanto o servidor como os clientes. Por este motivo, o impacto no desempenho do computador foi na verdade superior ao de condições normais, ainda que largamente imperceptíveis na máquina em questão.

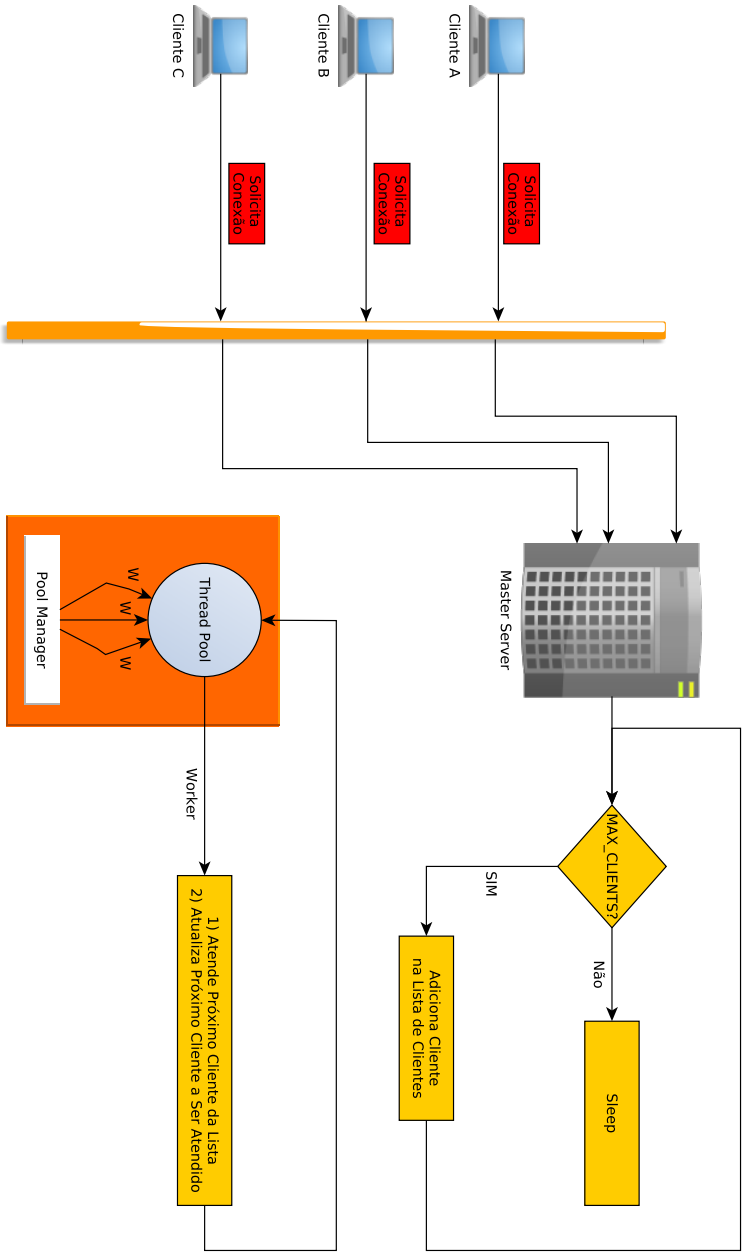
5 Conclusão

Ao longo deste projecto conseguimos imlpementar uma solução funcional para um esquema de cliente - servidor.

Ainda que com arestas a limar, este servidor em particular, que foi sem dúvida onde investimos mais tempo, suporta um número de clientes que é apenas limitado pela máquina em questão. Não só cumpre os requisitvos definidos como os supera nalguns casos.

Na implementação, o servidor consegue lançar um número dinâmico de *threads* que multiplexa entre si os muitos clientes que possam estar disponíveis para executar. Enquanto que esta implementação trás versatilidade ao servidor, introduz também o problema acrescido de não aproveitar o bloqueio típico de operações de *IO* para evitar esperas activas. Ainda assim, mesmo não tendo tido tempo de as implementar, discutimos alternativas reais que poderiam minimizar ou eliminar este problema, levando a um servidor bastante robusto.

A Diagrama do modelo implementado


Cliente C
Cliente B
Cliente A
Solicita Conexão
Solicita Conexão
Solicita Conexão
Pool Manager
W
W
W
Thread Pool
Master Server
Worker
MAX_CLIENTS?
SIM
Não
1) Atende Próximo Cliente da Lista
2) Atualiza Próximo Cliente a Ser Atendido
Adiciona Cliente na Lista de Clientes
Sleep

B Manual do cliente

```
************************************************************
Y A S C  --  ClientHelpFile   [Revision 2]
************************************************************

=== Calling Procedure ===

  ./yascC.exe <hostname> <port>
    It is suggested to run from within directory ./program/.
    Use 'make␣runC' after changing the arguments to avoid writing the
        arguments all the time.

  Options:

  -g              start in debug mode

  -f <file.txt>   takes commands from file.txt instead of command line
                  (.txt extension not enforced)

  -l              creates log.txt and runs silently
                  (Critical errors are still posted on the command line
                      rather than the log file)


=== Syntax ===

  Interpreter:
    The parsing function only takes into account blocks of characters
        separated by white space.

  Numbers:
    1) have to be integers
    2) shall not be preceded by 'D'
    3) may have an extra attribute, the signal, in the format '-###' or '
        +###' without spaces
    4) may be given in decimal format as '###'; hexadecimal as '0x###' or
        octal as '0###'

  Math:
    Server uses Reverse Polish Notation.
    Any inconformity will result in unwanted behaviour, with or without
        errors, depending on mathematical correctness.

  Special cases:
    ';' marks EVERYTHING written afterwards in the same line as comments to
         be discarded.

    In a command file, every command is issued as soon as it is read, so
        EXIT marks the end of the program
    regardless of any command or random text that comes afterwards.
```

```
=== Commands ===

  'I'                    Begins session: connection is established; server
     issues a stack.
                         Server cleans the stack if there is one already.

  'K'                    Ends session: server erases stack; connection is
     terminated. Returns nothing.

  'P'                    Asks server for the stack size. Always succeeds.

  'T'                    Asks server for the top most value in the stack.

  'R'                    Asks server for end result; cleans the stack only
     if there is one value in the stack.
                         If the result is a partial one (i.e. there are
                            still values in the stack), or if the stack is
                            empty,
                         the server shall return an error. To see partial
                            results use 'T'.
                         In case of error, returns BIG_STACK or BAD_STACK.

  'G'                    Sets debug flag for extra output.

  '+','-','*','/','%'   Integer basic operations. In case of error, return
     OUT_OF_RANGE, and also DIV_0 for '/' and '%'.

  'D' [deprecated]       Command is still issued internally whenever a
     number is found; direct usage results in parsing error.
                         Doesn't␣break␣the␣flow␣of␣the␣program,␣nor␣it␣is␣
                            checked␣for␣good␣use␣(i.e.␣right␣before␣a␣
                            number).

␣␣'HELP'␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Opens␣*this*␣document␣on␣the␣standard␣output,␣even␣
    if␣the␣-l␣flag␣is␣set.

␣␣'EXIT'␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Exits␣the␣program␣without␣prompt.


===␣Output␣===

␣␣There␣are␣three␣kinds␣of␣output:

␣␣␣␣>>␣␣␣␣␣␣␣␣Internal␣errors␣that␣break␣functionality␣of␣the␣client;␣doesn'
    t get redirected or suppressed with any flag.
             Program always exits after one of these.

    ::       Normal output; includes minor internal errors, and external
       errors (i.e. server errors); gets redirected with -l flag.
```

```
    DEBUG:   Additional / alternate output; present when -g flag is set or
        'G' issued; gets redirected with -l flag.


=== ERROR codes ===

  These codes are defined in globalHeader.h.

  OUT_OF_RANGE      Over/Underflow; arithmetic result is out of range (min/
     max range defined as INT_MIN/INT_MAX in limits.h).
  DIV_0             Division by 0 detected.
  BIG_STACK         Stack is bigger than expected for specified operation.
  BAD_STACK         Not enough operands for specified action.



(SCROLL UP ^)
************************************************************
```

C Manual do servidor

```
************************************************************
Y A S C  --  ServerHelpFile   [Revision 1]
************************************************************

=== Calling Procedure ===

  ./yascS.exe <port>
    It is suggested to run from within directory ./program/.
    Use 'make␣runS' after changing the arguments to avoid writing the
        arguments all the time.

  Options:

  -v            verbose mode


=== Syntax ===

  Interpreter:
    The parsing function only takes into account commands separated by
        white space.


=== Commands ===

  'M'                   Prints information about every client logged on the
      server.
                        Format is: <IP address> [ stack element 1, stack
                            element 2, ... ].

  'V'                   Verbose mode.

  'HELP'                Opens *this* document on the standard output.

  'F'                   Exits the program without prompt.


=== Services ===

  All communications shall use a package of 9 bytes.
  The first one is a message: 'V', 'E' or 'I'.
  The second one is an integer in hexadecimal format.


  Calculator:

    This server deploys a threaded fully compliant integer calculator with
        correct handling of mathematical / computational exceptions.
```

```
    For each correctly handled command, it shall return a message 'V' and a
        number, being 0 for padding, and a numerical result for relevant
       commands.

    The calculator interface is possible through a socket in the following
       format:

      'I'                   Begins session: connection is established and a
          stack is issued.
                            The server cleans the stack if there is one
                               already.

      'K'                   Ends session: stack is erased; connection is
         terminated. Returns nothing to the client.

      'P'                   Returns the stack size. Always succeeds.

      'T'                   Returns the top most value in the stack.

      'R'                   Returns end result and cleans the stack only if
          there is one value in the stack.
                            If the result is a partial one (i.e. there are
                               still values in the stack), or if the stack
                                is empty,
                            the server shall return an error. To see
                               partial results use 'T'!
                            In case of error, returns BIG_STACK or
                               BAD_STACK.

      '+','-','*','/','%'   Integer basic operations. In case of error,
         returns OUT_OF_RANGE, and also DIV_0 for '/' and '%'.

      'D␣hex␣number'        Hexadecimal number is converted to integer and
         stored in the client's␣stack.


␣␣Error␣codes:

␣␣␣␣The␣server␣shall␣return␣these␣codes,␣defined␣in␣globalHeader.h,␣
    alongside␣with␣a␣message␣'E'␣in␣case␣of␣ill␣usage:

␣␣␣␣BAD_CMD␣␣␣␣␣␣␣␣␣␣␣␣Nonconforming␣message␣received.
␣␣␣␣OUT_OF_RANGE␣␣␣␣␣␣␣Over/Underflow;␣arithmetic␣result␣is␣out␣of␣range␣(
    min/max␣range␣defined␣as␣INT_MIN/INT_MAX␣in␣limits.h).
␣␣␣␣DIV_0␣␣␣␣␣␣␣␣␣␣␣␣␣␣Division␣by␣0␣detected.
␣␣␣␣BIG_STACK␣␣␣␣␣␣␣␣␣␣Stack␣is␣bigger␣than␣expected␣for␣specified␣operation
    .
␣␣␣␣BAD_STACK␣␣␣␣␣␣␣␣␣␣Not␣enough␣operands␣for␣specified␣action.

␣␣␣␣The␣server␣shall␣return␣these␣codes,␣defined␣in␣globalHeader.h,␣
    alongside␣with␣a␣message␣'I'␣in␣case␣of␣internal␣error:
```

```
␣␣␣␣[none␣defined]


===␣Output␣===

␣␣There␣are␣three␣kinds␣of␣output:

␣␣␣␣>>␣␣␣␣␣␣␣␣Internal␣errors␣that␣break␣functionality␣of␣the␣server.␣
    Program␣always␣exits␣after␣one␣of␣these.

␣␣␣␣::␣␣␣␣␣␣␣␣Normal␣output;␣includes␣minor␣internal␣errors.

␣␣␣␣␣␣␣␣␣␣␣␣␣␣'M'␣command␣has␣a␣specific␣printing␣layout␣already␣covered␣
    above.


(SCROLL␣UP␣^)
************************************************************
```