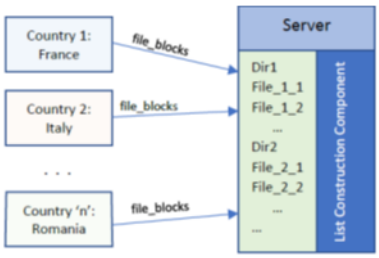
Documentatie P1 PPD

Analiza cerintelor

Concurs international

Consideram extinderea problemei de la laboratorul 5 la o aplicatie client-server.
Fiecare tara (client) trimite catre organizator (server) rezultatele propriilor concurenti, pe care le consideram existente in fisiere.
Trimiterea se face in calupuri de cate 20 perechi (ID_concurent, punctaj) la un interval de Δx (pentru a se simula actiunea userului pe interfata).
Suplimentar pentru 2 puncte varianta 2 de transmitere: clientii trimit fisierele cu rezultatele pentru o problema folosind o impartire in blocuri de dimensiune predefinita (file_blocks of size bytes). La nivelul serverului dupa ce s-au primit toate blocurile se reformeaza fisierul si apoi se continua ca si la laboratorul 5. Pentru obtinerea celor 2 puncte suplimentare este necesar sa se implementeze ambele variante.
Serverul preia datele de la clienti si adauga la perechile trimise de clienti ID de tara creand triplete (ID_tara, ID_concurent, punctaj) si le adauga intr-o coada similara celei realizate la laboratorul 5.
Lista finala se actualizeaza tot cu operatiile corepunzatoare descrise la Laborator 5. Lista nu trebuie sa fie ordonata dupa fiecare inserare ci este suficienta o sortare la final.
Serverul se foloseste de p_r threaduri pentru a prelua aceste date de la clienti (thread pool cu p_r threaduri) si p_w threaduri care adauga in lista globala de concurenti.
Dupa trimiterea datelor corespunzatoare fiecarei probleme, fiecare client (tara) trimite o cerere de informare referitoare la clasamentul tarilor.
Punctajul unei tari este egal cu suma punctajelor tuturor concurenților din acea tara.
La primirea unei astfel de cereri serverul (va crea un future) va începe sa calculeze anterior acest clasament si atunci cand finalizeaza va trimite raspunsul catre client. Daca serverul are clasamentul calculat pe tari la un interval de timp mai mic decat un Δt dat atunci nu mai reface calculul si trimite acel clasament – aceste calcule se fac in main-thread-ul corespunzator serverului.
La final, fiecare client trimite o cerere pentru a primi rezultatul final.
Dupa ce finalizeaza clasamentul final serverul salveaza intr-un fisier clasamentul final pe concurenti si in alt fisier clasamentul pe tari.
Apoi trimite continutul acestor fisiere la clienti ca si raspuns la ultima cerere a acestora.



Obiectiv:

1. Implementarea unei aplicatii client-server
2. Folosirea executiei concurente prin apeluri asincrone
3. Folosirea mecanismelor: future/promises si thread_pool
4. Analiza imbunatatirii performantei executiei unei aplicatii (de tip business) prin programare concurenta

Proiectare

Ca si tip abstract de date vom folosi o lista simplu inlantuita, thread-safe, cu sentinele, neordonata. Lista va respecta fine-grained synchronization adica vom bloca cat mai putin necesar pentru a face o operatie. Pentru procesarea intrarilor vom folosi un tip abstract de date de tip coada thread-safe cu capacitate maxima, iar sincronizarea o vom face cu ajutorul variabilelor conditionale.
Numarul de threaduri (configuratia) se va transmite prin intermediul parametrilor.
Serverul va folosi un thread-pool pentru gestionarea conexiunilor de la clienti, fiecarui client fiindu-i asignat cate un thread din pool.

Detalii de implementare

```
static
VOID
ProducerWorker(_In_ int ThreadIndex)
{
    try
    {
        ConcursClient concursClient;
        bool status = concursClient.Connect();
        if (!status)
        {
            return;
        }

        concursClient.Send(ThreadIndex);

#ifndef V2
        std::vector<Participant> participantsChunk;
        std::vector<std::vector<Participant>> participants;
#endif

        for (int j = 1; j <= TASKS_COUNT; ++j)
        {
            CHAR filePath[MAX_PATH] = { 0 };
            _snprintf_s(filePath, MAX_PATH, _TRUNCATE, "D:\\facultate-repo\\sem5\\ppd\\p1\\inputs\\Rezult

            std::ifstream fin(filePath, std::ios::binary);
            if (!fin.is_open())
            {
                continue;
            }

#ifdef V2
            CHAR buffer[BLOCK_SIZE] = { 0 };
            while (!fin.eof())
            {
                fin.read(buffer, BLOCK_SIZE);
                std::streamsize bytesRead = fin.gcount();

                if (bytesRead == 0)
                {
                    break;
                }

                concursClient.Send(buffer, static_cast<int>(bytesRead));
            }
#else
            int participantId, participantScore;
            while (fin >> participantId >> participantScore)
            {
                Participant participant;
                participant.SetId(participantId);
                participant.SetScore(participantScore);

                participantsChunk.emplace_back(participant);
                if (participantsChunk.size() == CHUNK_MAX_SIZE)
                {
                    participants.emplace_back(participantsChunk);
                    participantsChunk.clear();
                }
            }
```

```
#endif

            fin.close();

#ifndef V2
            if (!participantsChunk.empty())
            {
                participants.emplace_back(participantsChunk);
                participantsChunk.clear();
            }

            for (const std::vector<Participant>& chunk : participants)
            {
                concursClient.Send(chunk);
                std::this_thread::sleep_for(std::chrono::milliseconds(gDeltaX));
            }
            participants.clear();
#endif

            RequestPartialRankings(concursClient);
        }

        RequestFinalRankings(concursClient);

        concursClient.Disconnect();
    }
    catch (const std::exception& exception)
    {
        std::cout << exception.what();
    }
}
```

`ProducerWorker` este functia pe care o folosesc workerii-client (thread-urile) ce citesc din fisier.
Pentru V1, se citeste cate un participant si se acumuleaza intr-un vector-calup. Cand acest vector atinge o dimensiune specifica (in cazul nostru 20), se vor trimtie toti acesti participanti catre server pentru a fi procesati. Pentru V2, se citesc un numar specific de bytes din fisier si se trimit cate server pana se citeste tot fisierul. Serverul va extrage informatiile necesare punand cap la cap toate aceste calupuri de bytes. Pentru ambele variante, dupa ce fisierul a fost procesat in intregime este trimisa catre server o cerere pentru clasamentul provizoriu. La final, dupa ce toate fisierele au fost procesate, este trimisa catre server o cerere pentru clasamentul final.

```
VOID
ConcursServer::HandleConnections()
{
    int connections = 0;
    while (connections < this->clientProducersCount)
    {
        SOCKADDR_IN clientInfo = { 0 };
        int clientInfoSize = sizeof(clientInfo);
        SOCKET* clientSocket = new SOCKET(accept(this->listenSocket, reinterpret_cast<PSOCKADDR>(&clientI
        if (*clientSocket == INVALID_SOCKET)
        {
            std::cout << "accept failed with status " << WSAGetLastError() << std::endl;
            closesocket(*clientSocket);
            delete clientSocket;
        }
        else
        {
            ++connections;
            this->threadPool->detach_task([&, clientSocket, clientInfo]
```

```
            {
                CHAR clientIP[INET_ADDRSTRLEN] = { 0 };
                inet_ntop(AF_INET, &clientInfo.sin_addr, clientIP, INET_ADDRSTRLEN);

                std::cout << "Client connected from IP: " << clientIP << std::endl;

                PCLIENT_CONTEXT clientContext = new CLIENT_CONTEXT{ .Socket = *clientSocket, .IPv4 = clie
                this->InsertNewClient(clientSocket, clientContext);
                this->HandleConnection(*clientContext);
            });
        }
    }
}
```

`HandleConnections` este functia apelata pentru a accepta conexiunile clientilor. Pentru fiecare client, este asignat un thread din thread-pool care va procesa informatiile primite de la acesta.

```
VOID
ConcursServer::HandleConnection(_Inout_ CLIENT_CONTEXT& ClientContext)
{
    int status = 0;
    do
    {
        CHAR buffer[DEFAULT_BUFLEN] = { 0 };
        int bufferSize = sizeof(buffer);

        status = recv(ClientContext.Socket, buffer, bufferSize, 0);
        if (status > 0)
        {
            std::string data(buffer, status);
            this->incompleteDataMap[ClientContext.Socket] += data;

#ifdef V2
            if (this->incompleteDataMap[ClientContext.Socket].ends_with("99999\r\n"))
            {
                this->ProcessData(this->incompleteDataMap[ClientContext.Socket], ClientContext);
                this->incompleteDataMap[ClientContext.Socket].erase();
                this->pcQueue.UnregisterProducer();
                ClientContext.NotSendingAnymore = true;
            }
            else if (this->incompleteDataMap[ClientContext.Socket].ends_with("99998\r\n"))
            {
                this->ProcessData(this->incompleteDataMap[ClientContext.Socket], ClientContext);
                std::thread([&]()
                {
                    std::promise<std::vector<std::pair<int, int>>> promise;
                    std::future<std::vector<std::pair<int, int>>> futureRanking = promise.get_future();

                    this->GetRankingManager().AddTask(promise);

                    const std::vector<std::pair<int, int>>& countryRanking = futureRanking.get();
                    this->Send(ClientContext.Socket, countryRanking);
                }).detach();
                this->incompleteDataMap[ClientContext.Socket].erase();
            }
#else
            size_t newlinePos;
            while ((newlinePos = this->incompleteDataMap[ClientContext.Socket].find("\r\n")) != std::str
            {
                std::string completeData = this->incompleteDataMap[ClientContext.Socket].substr(0, newli
```

```
                this->ProcessData(completeData, ClientContext);

                this->incompleteDataMap[ClientContext.Socket].erase(0, newlinePos + 2);
            }
#endif
        }
        else if (!status)
        {
            std::cout << "Connection closing..." << std::endl;
            this->incompleteDataMap.erase(ClientContext.Socket);
        }
        else
        {
            std::cout << "recv failed: " << WSAGetLastError() << std::endl;
            this->incompleteDataMap.erase(ClientContext.Socket);
            return;
        }
    } while (status > 0 && !ClientContext.NotSendingAnymore);
}

VOID
ConcursServer::ProcessData(_In_ const std::string& Data, _Inout_ CLIENT_CONTEXT& ClientContext)
{
    if (!Data.ends_with("\r\n"))
    {
        return;
    }

#ifdef V2
    std::stringstream ss(Data);
    std::string token;
    while (std::getline(ss, token, '\r'))
    {
        if (ss.peek() == '\n')
        {
            ss.ignore();
        }

        std::stringstream tokenStream(token);
        int field1 = -2, field2 = -2;
        tokenStream >> field1 >> field2;

        if (field1 != -2 && field2 == -2 && field1 < 6)
        {
            ClientContext.Country = field1;
        }
        else if (field1 != -2 && field2 != -2)
        {
            if (!ClientContext.Country)
            {
                std::cout << "Logical error: ClientContext.Country is null" << std::endl;
                __debugbreak();
            }

            Participant participant;
            participant.SetId(field1);
            participant.SetScore(field2);
            participant.SetCountry(ClientContext.Country);

            this->pcQueue.Produce(participant);
        }
    }
```

```
#else
    const std::string& processedData = Data.substr(0, Data.size() - 2);

    if (!Data.contains(" "))
    {
        int data = std::stoi(processedData);
        if (data == 99998) // partial leaderboard request
        {
            std::thread([&]()
            {
                std::promise<std::vector<std::pair<int, int>>> promise;
                std::future<std::vector<std::pair<int, int>>> futureRanking = promise.get_future();

                this->GetRankingManager().AddTask(promise);

                const std::vector<std::pair<int, int>>& countryRanking = futureRanking.get();
                this->Send(ClientContext.Socket, countryRanking);
            }).detach();
        }
        else if (data == 99999) // final leaderboard request
        {
            this->pcQueue.UnregisterProducer();
            ClientContext.NotSendingAnymore = true;
        }
        else
        {
            ClientContext.Country = data;
        }
    }
    else
    {
        if (!ClientContext.Country)
        {
            std::cout << "Logical error: ClientContext.Country is null" << std::endl;
            __debugbreak();
        }

        Participant participant = Participant::Deserialize(processedData);
        participant.SetCountry(ClientContext.Country);

        this->pcQueue.Produce(participant);
    }
#endif
}
```

Pentru V1, fiecare participant va fi procesat individual, iar pentru V2 se acumuleaza informatie pana cand s-a citit tot fisierul, iar apoi este procesata informatia si sunt extrasi participantii.
Procesarea participantilor pe server se intampla asemanator ca si la laboratorul 5.

Testare si analiza performantei

C++

Numar produceri (p_r)	Numar consumeri (p_w)	Delta T	Delta X	Timp executie (secunde)
4	4	1ms		2.11105
		2ms		1.68640

		4ms	12ms	1.69083
2	2	1ms		2.41733
		2ms		2.41816
		4ms		2.57977
4	2	1ms		2.05403
		2ms		2.06549
		4ms		1.98636
4	8	1ms		1.83080
		2ms		1.68035
		4ms		1.68318

Concluzii

Putem observa ca cu cat mai multi consumatori avem, cu atat si timpul de executie s-a diminuat. De asemenea, se poate observa si ca numarul de produceri influenteaza semnificativ timpul de executie, cu cat numarul acestora este mai mare, cu atat si timpul de executie este mai favorabil. Mai mult decat atat, o diferenta notabila este cand se foloseste Δt cu valori de 1ms si 2ms, pare ca atunci cand Δt este mai mare si nu se mai recalculeaza clasamentele, timpul de executie se diminueaza. Abordarea folosita se dovedeste a fi eficienta in contextul problemei noastre.