INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

DEPARTAMENTO DE ENGENHARIA DE ELETRÓNICA E
TELECOMUNICAÇÕES E DE COMPUTADORES

Sistemas de Informação

PergNET - Sistema de apoio ao estudo através da realização de jogos didácticos

Jorge Manuel Vital dos Santos
(Bacharel em Engenharia Informática e de Computadores)

TRABALHO DE PROJECTO PARA OBTENÇÃO DO GRAU DE MESTRE
EM ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Orientadores:
Mestre Nuno António Afonso Cunha Oliveira
Mestre Nuno Miguel Soares Datia

Júri:
Presidente: Mestre Vitor Almeida
Vogais:
Doutor Porfírio Pena Filipe
Mestre Nuno António Afonso Cunha de Oliveira
Mestre Nuno Miguel Soares Datia

Junho de 2010

Resumo

Uma forma suplementar de apoio ao estudo do aluno, numa disciplina, será o de permitir que os alunos testem os conceitos apreendidos durante o seu processo de estudo pessoal através de jogos didácticos. Estes jogos, realizados individualmente ou em grupo, apresentam um conjunto de perguntas de desafio onde cada interveniente irá seleccionar uma resposta. Esta forma de realização de um questionário aparece aliada à componente lúdica de um jogo que possibilita a aprendizagem progressiva e incremental com o valor acrescentado de permitir aos intervenientes a discussão sobre o tópico da pergunta, sobre as respostas e sobre outros aspectos relacionados com a temática consolidando o seu conhecimento.

O desenvolvimento de jogos de uma forma distribuída apresenta alguns desafios, nomeadamente, a sincronização entre os diversos intervenientes e a ultrapassagem de dificuldades inseridas por *firewalls* e NATs. Estes problemas devem ser resolvidos por uma camada de comunicação que suporte o desenvolvimento de aplicações distribuídas.

O objectivo principal deste trabalho foi a criação desse sistema de comunicações baseado na tecnologia .NET, que permite interligar vários utilizadores localizados em redes privadas ou pública. A infra-estrutura possibilita a constituição de diversos grupos de participantes sendo possível o envio de mensagens ponto a ponto e a difusão de mensagens para o grupo.

A avaliação da infra-estrutura de comunicações foi realizada através de uma aplicação de apoio ao estudo dos alunos por meio de jogos didácticos, mostrando que os problemas enunciados foram resolvidos.

Palavras-Chave

NAT, sistemas distribuídos, grupos de comunicação, jogos didácticos.

Abstract

One additional way to allow students to test their knowledge about school subjects is through educational games. Some authors suggested that computer games increase learner's motivation, and motivation is an essential component of any learning activity.
Those games can be performed with one single player or several players at the same time; it presents a set of challenging questions where each player will select an answer. This type of quiz joins the playful and educational components allowing the students to monitor their evolution.

Game developing in a distributed way presents some challenges, namely, several players synchronisation and overcoming some difficulties inserted by *firewalls* and NATs. These problems should be solved using a communication layer that support distributed applications development.

These work main purpose was the design of a system communication based on .NET technology, which allows connection between several users located in various public or private nets. This structure allows the organization of various groups and also making possible to send messages end-to-end and to the group.

The communications infrastructure evaluation was preformed by means of an educational game.

Key-words

NAT, distributed systems, communication groups, educational games.

Índice

Glossário ... 1
1 Introdução ... 3
1.1 Motivações ... 3
1.2 Descrição do Problema ... 3
1.3 Estrutura do Relatório ... 5
2 Estado da Arte ... 7
2.1 Conjunto de Técnicas ... 7
2.1.1 *Relaying* ... 7
2.1.2 *Hole Punching* com UDP do NAT ... 8
2.1.3 *Hole Punching* com TCP do NAT ... 12
2.2 Software ... 15
2.2.1 JXTA ... 15
2.2.2 Skype ... 16
2.2.3 MSN Messenger ... 17
3 Descrição da Solução ... 19
3.1 Arquitectura da Solução ... 19
3.2 Comunicações de Grupos ... 22
3.3 Módulos Lógicos da Solução ... 23
3.4 Serviços Disponibilizados pela Plataforma de Comunicações ... 27
3.4.1 Serviços de iniciação e finalização ... 27
3.4.2 Serviços de notificação (ocorrem em determinados eventos) ... 28
3.4.3 Serviços que podem ser solicitados pela aplicação ... 30
3.5 Codificação de um serviço ... 32
3.5.1 Natureza das Mensagens usadas pelos serviços ... 32
3.5.2 Controlador da Mensagem ... 35
3.5.3 Mensagem Base ... 40
3.6 Camadas de Software ... 43
3.6.1 Camada XTCPClient ... 44
3.6.2 Camada XTCPServer ... 47
3.7 Servidor de Comunicações ... 50
4 Teste da Solução ... 53
4.1 O Jogo Como Teste da Solução ... 54
4.1.1 O Desenrolar de Um Jogo ... 54

5 Conclusões ... 67
5.1 Plataforma de Comunicações Desenvolvida ... 67
5.2 Trabalho Futuro ... 67
6 Referências Bibliográficas ... 69
Anexo A – Modelo de Dados do Jogo ... 71
A.1 Modelo Entidade-Associação ... 71
A.2 Modelo Relacional ... 72
Anexo B - Interacção do jogo com o seu sistema de informação ... 73
Anexo C - Utilizadores do Jogo ... 79
Anexo D - Ambiente de Utilização do Jogo ... 81
Anexo E - Ficheiros que compõem a solução ... 83
Anexo F – Resumo dos serviços de iniciação e finalização ... 85
Anexo G – Resumo dos serviços de notificação ... 87
Anexo H – Resumo dos serviços que podem ser solicitados directamente pela aplicação ... 89

Índice de Ilustrações

Ilustração 1 - Domínios de endereços IP privados e públicos que compõem a Internet 4
Ilustração 2 - Atravessamento do NAT por *Relaying* 8
Ilustração 3 - Antes do *Hole Punching* com UDP 10
Ilustração 4 - *Hole Punching* com UDP propriamente dito 10
Ilustração 5 - Após *Hole Punching* com UDP 11
Ilustração 6 - *Sockets* e portos necessários em cada nó para o *Hole Punching* por TCP 14
Ilustração 7 - Relacionamento entre os hospedeiros comuns, os super-nós e o servidor de login. 17
Ilustração 8 - Arquitectura adoptada para as comunicações 20
Ilustração 9 - Arquitectura Física da Solução 21
Ilustração 10 - Exemplo de uma disposição de grupos conforme as necessidades da aplicação num determinado instante 23
Ilustração 11 - Relação entre o software da aplicação consumidora e da plataforma de comunicações 24
Ilustração 12 - Diagrama de sequência para um pedido com retorno de dados e sem ocorrência de *timeout* 34
Ilustração 13 - Diagrama de sequência para um pedido com retorno de dados e com ocorrência de *timeout* 35
Ilustração 14 - Diagrama de sequência para o controlador CMsgCtrl_GetLocalEndpoint (que devolve a lista dos *endpoints* locais dos destinatários) 39
Ilustração 15 - Formato da informação trocada pelos intervenientes 39
Ilustração 16 - Diagrama UML que evidencia a relação entre as interfaces IMsgCtrl e IMsg e os controladores 43
Ilustração 17 - Interacção entre as camadas de software do sistema de comunicações 44
Ilustração 18 - Estrutura da Tabela de Pedidos localizada no cliente 45
Ilustração 19 - Estrutura da Tabela de Mensagens Recebidas para Tratamento FIFO 46
Ilustração 20 - Exemplo de como se pode encontrar a Tabela de Mensagens Recebidas para Tratamento FIFO 47
Ilustração 21 - Estrutura da Tabela de Contadores das Mensagens FIFO a enviar 47
Ilustração 22 - Tabela de Grupos 48
Ilustração 23 - Tabela de Clientes 49
Ilustração 24 - Tabela de Pedidos do Servidor 49
Ilustração 25 - Servidor de comunicações usando uma *Thread Pool* 50

Ilustração 26 - Relação da Arquitectura Física e Lógica do Jogo.. 54
Ilustração 27 - Ecrã de Configuração do Jogo Simples 56
Ilustração 28 - Ecrã de indicação de que se está a localizar jogadores disponíveis................ 57
Ilustração 29 - Ecrã com indicação dos jogadores disponíveis... 57
Ilustração 30 - Ecrã com indicação de que o convite está a ser feito (computador de Jorge Santos).. 58
Ilustração 31 - Ecrã com indicação do convite (computador de Rui Tavares) 58
Ilustração 32 - Lista de jogadores seleccionados para o jogo (computador do Jorge Santos). 59
Ilustração 33 - Indicação de que o jogo vai começar (computador do Rui Tavares).............. 59
Ilustração 34 - Ecrã a dar indicação de quem irá jogar de seguida. Nota o botão “Avançar” só aparece no jogador activo. 60
Ilustração 35 - Ecrã em que o jogador activo está a responder a uma questão de escolha múltipla (computador do Jorge) 61
Ilustração 36 - Ecrã em que o jogador passivo assiste a uma resposta de escolha múltipla (computador do Rui). A estrela indica a posição do rato no ecrã do jogador activo. 61
Ilustração 37 - Ecrã obtido após selecção da resposta correcta 62
Ilustração 38 - Exemplo de um ecrã em que a resposta não é de escolha múltipla. 63
Ilustração 39 - Ecrã que o jogador passivo recebe para classificar a resposta do jogador activo 64
Ilustração 40 - Ecrã que o jogador activo visualiza após responder a uma pergunta de desenvolvimento 64
Ilustração 41 - Ecrã com as classificações atribuídas à resposta de desenvolvimento. 65
Ilustração 42 - Ecrã com a classificação final... 66
Ilustração 43 - Formulário para inserir um novo utilizador.. 73
Ilustração 44 - Formulário para inserir classes no sistema 74
Ilustração 45 - Formulário para inserir subclasses no sistema.. 75
Ilustração 46 - Formulário para inserir perguntas no sistema... 75
Ilustração 47 - Formulário para consulta de perguntas existentes na Base de Dados............. 76
Ilustração 48 - Formulário de uma pergunta previamente inserida e passível de ser aprovada. 77
Ilustração 49 - Contexto de Utilização.. 81

Índice de Tabelas

Tabela 1 - Resumo dos Utilizadores 79
Tabela 2 - Ficheiros que compõem a solução 83
Tabela 3 - Serviços de iniciação e finalização solicitados pelo servidor da aplicação consumidora 85
Tabela 4 - Serviços de iniciação e finalização solicitados pelo cliente da aplicação consumidora 85
Tabela 5 - Serviços de notificação (ocorrem em determinados eventos) 87
Tabela 6 – Serviços que podem ser solicitados pelo cliente da aplicação 89
Tabela 7 – Serviços que podem ser solicitados pelo servidor da aplicação 90

Índice de Códigos

Código 1 - Interface do Controlador de Mensagem 36
Código 2 - Controlador "GetLocalEndpoint" 38
Código 3 - Interface "IMsg" 40

Glossário

Delegate — Tipo que referencia um método. Após o *delegate* ser associado a um método, comporta-se exactamente como esse método. O método *delegate* pode ser usado como qualquer outro método, com parâmetros e com um valor de retorno.

Endpoint — Endereço composto pelo par IP/Porto.

Firewall — Barreira desenhada para impedir comunicações, cujo acesso não é autorizado ou não é desejado, entre duas zonas de uma rede.

Hole Punching — Conjunto de técnicas empregues por aplicações que pretendem atravessar o NAT.

Hop — "Salto" dado por um pacote, entre um nó (router, computador ou dispositivos) e outro nó da rede, antes de chegar ao seu destino.

NAT — (*Network Address Translation*) É o processo de conversão de um endereço de rede, aplicado a um pacote IP, noutro endereço de rede, enquanto o pacote atravessa um dispositivo de *routing*, com o propósito de transformar um espaço de endereçamento noutro espaço de endereçamento.

NAT *Oubound* — Tipo de NAT que permite, por omissão, a criação de sessões de saída para atravessar o NAT, e impede a recepção de pacotes de entrada excepto quando estes fazem parte de uma sessão iniciada a partir de dentro da rede privada, ou seja quando são uma “resposta”.

Nó — Dispositivo ligado a uma rede de computadores, tais como, um computador ou um router.

Payload — Carga útil em protocolos de comunicação. Refere-se ao dado real que está a ser transmitido. O *payload* é, normalmente, precedido de um

cabeçalho que identifica o transmissor e o receptor dos dados. O *payload* é descartado quando chega ao destinatário.

Peer — Nó com função simultânea de fornecedor e consumidor de recursos numa rede. As redes a que pertencem estes nós designam-se por redes *peer-to-peer* por oposição às redes cliente-servidor cujos nós ou tem papéis de servidor ou de cliente.

PSTN — (*Public Switched Telephone Network*) Rede pública de telefonia comutada é o termo usado para identificar a rede telefónica mundial comutada por circuitos destinada ao serviço telefónico. Inicialmente foi projectada como uma rede de linhas fixas e analógicas. Actualmente é digital e inclui também dispositivos móveis como os telemóveis.

Relay — Nó com função de retransmitir dados.

Relaying — Retransmissão da informação.

Rendezvous Server — Servidor que funciona como ponto de encontro entre clientes.

TCP — (*Transmission Control Protocol*) Um dos principais protocolos do IP que é fiável (tem controlo de erros), garante a entrega e fornece os pacotes de forma a respeitar a ordem da informação enviada pelo emissor. É um protocolo orientado a fluxos de bytes.

UDP — (*User Datagram Protocol*) Um dos principais protocolos do IP, da camada de transporte, que não é fiável (não tem controlo de erros), nem garante a entrega. Fornece um serviço sem conexão (não mantém um relacionamento longo com o cliente).

VoIP — Voz sobre IP. É o *routing* de conversação humana usando a Internet ou qualquer outra rede de computadores baseada no Protocolo de Internet, tornando a transmissão de voz noutro serviço suportado pela rede de dados.

1 Introdução

1.1 Motivações

As motivações que estiveram na base do trabalho foram o desenvolvimento de um sistema de comunicações que permitisse ligar utilizadores localizados em diversas redes privadas ou pública e o aprofundamento dos conhecimentos na *Framework .NET* relacionado com a comunicação entre processos.

1.2 Descrição do Problema

Um sistema de comunicações para ser bem sucedido precisa que os seus intervenientes consigam comunicar entre si. Um dos problemas que se coloca é descobrir os endereços dos destinatários e outro problema é conseguir fazer chegar a mensagem ao receptor.

A determinação dos endereços dos possíveis destinatários pode ser resolvida, através do registo dos mesmos, numa base de dados central cujo local foi previamente convencionado, como acontece nas arquitecturas híbridas *Peer-to-Peer* [1].

O sucesso da entrega da mensagem ao destinatário está dependente de várias barreiras que possam existir na arquitectura a que pertencem as intervenientes. Acontece que a arquitectura da Internet de endereços originais uniforme, em que cada máquina possui um endereço exclusivo IP e que pode comunicar directamente com qualquer outro nó, foi substituída de facto por uma nova arquitectura de endereço de Internet. Esta nova arquitectura consiste num domínio de endereços global e muitos domínios de endereços privados interligados pela *Network Address Translator* (NAT) [2]. Um dispositivo NAT permite que vários nós privados comuniquem usando um conjunto de endereços IP (normalmente um) públicos e, dessa forma, resolver o problema da escassez de espaço de endereços IPv4 [2].

Nesta nova arquitectura, indicada na Ilustração 1, onde o NAT desempenha um papel crucial, a comunicação entre nós pertencentes ao mesmo domínio é fácil, pois basta conhecer o

endereço do destinatário referente ao domínio em causa para se conseguir fazer chegar uma mensagem. O envio de mensagens de uma máquina localizada num domínio privado para uma do domínio público, ou vice-versa, está dependente do NAT o que pode dificultar a sua entrega. Os nós das redes privadas podem, geralmente, abrir ligações TCP ou UDP para nós públicos. Quando isto acontece, os NATs apanhados no trajecto pelos pacotes dessas ligações, alocam *endpoints* públicos, embora temporários, para as ligações de saída (no sentido do domínio privado para o domínio público) e traduzem o endereço privado no *endpoint* criado. Outro comportamento típico de um NAT é bloquear o tráfego de entrada, ou seja, no sentido do domínio privado, salvo alguma configuração especial [3].

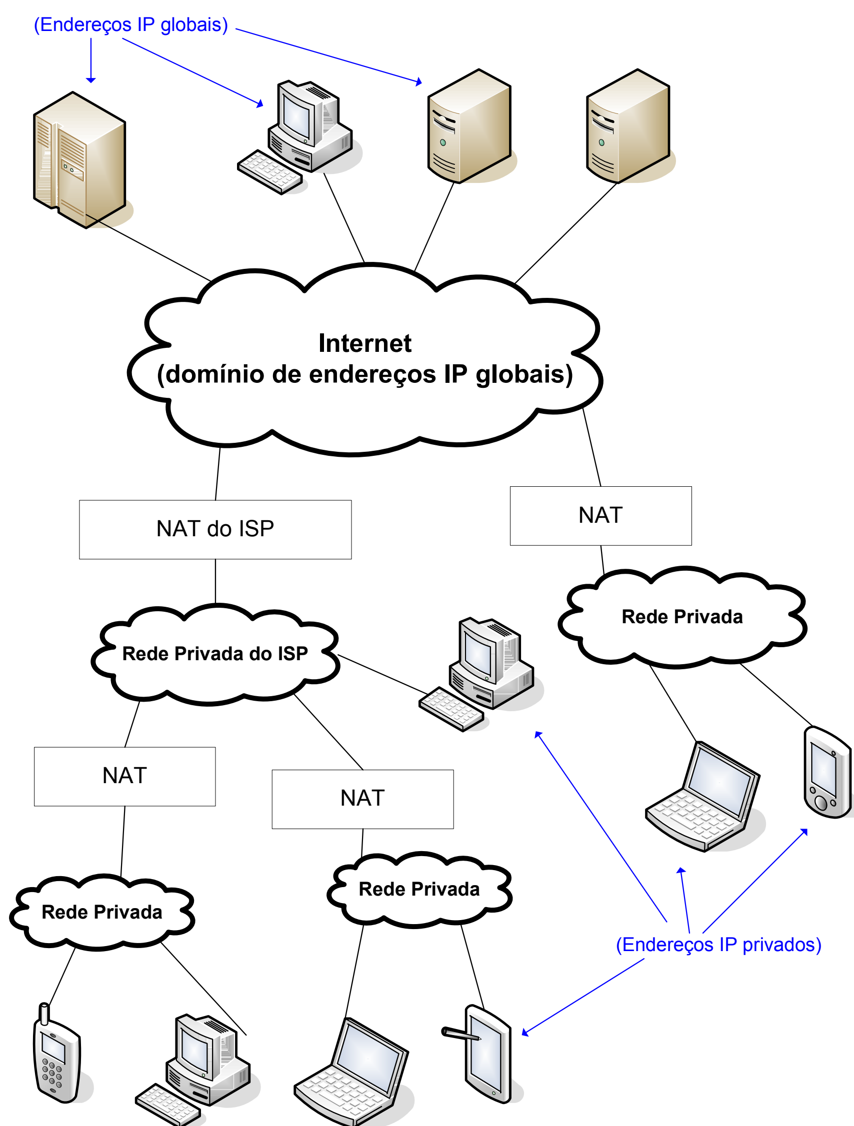


Ilustração 1 - Domínios de endereços IP privados e públicos que compõem a Internet

A função inicial do NAT é resolver a escassez de endereços do IPv4. Embora não seja uma solução a longo prazo, como previsto em 1994, a tecnologia NAT tende ainda a aumentar,

mesmo quando se tornar comum a utilização de endereços IPv6 [2]. Isto porque uma das formas mais fáceis de obter interoperabilidade entre as versões de IP é a utilização do NAT [2, 3]. Brian Carpenter estimou que 40% dos utilizadores encontram-se atrás de *firewalls* e/ou NATs [4].

O que se propõe fazer é elaborar um sistema de comunicações que permita os utilizadores identificarem os seus potenciais colaboradores activos no sistema e disponibilizar um meio virtual para a troca de informações entre si, mesmo quando estão localizados em redes distintas de domínio público ou privado. Para facilitar a troca de mensagens entre elementos cooperantes, o sistema irá permitir a criação de grupos de utilizadores. A entrega das mensagens *unicast* ou *multicast* terão de ser garantidas.

1.3 Estrutura do Relatório

O capítulo 1 descreve as motivações do projecto e faz uma descrição do problema.

No capítulo 2 apresenta-se o estado da arte no qual se focam um conjunto de técnicas para ultrapassar as dificuldades levantadas pela existência de mecanismos NAT entre as redes dos intervenientes. Apresenta-se, também, um conjunto de aplicações que fazem uso das técnicas mencionadas.

O capítulo 3 descreve a solução implementada. Este capítulo começa por explicar a arquitectura escolhida. Depois apresenta a forma como a plataforma de comunicações pretende tratar as comunicações de grupos. De seguida apresenta os módulos lógicos da solução e os serviços que o sistema de comunicações disponibiliza à sua aplicação consumidora. A seguir apresenta a forma como os serviços são codificados e como a plataforma de comunicações pode ser estendida no sentido de fornecer mais serviços. Depois são apresentadas as várias camadas de software que compõem o sistema de comunicações. Por fim descreve-se o servidor de comunicações.

No capítulo 4 descreve-se uma aplicação consumidora dos serviços prestados pelo sistema de comunicações. Esta aplicação foi concebida com o propósito de testar a plataforma de comunicações mostrando o quanto é vantajoso usar a plataforma na criação de sistemas distribuídos.

No capítulo 5 apresentam-se as conclusões do trabalho e trabalho futuro.

O capítulo 6 possui as referências bibliográficas consultadas de apoio à produção do trabalho.

2 Estado da Arte

Existem vários tipos de NAT, embora o mais comum seja o NAT *outbound*. Este tipo de NAT permite, por omissão, a criação de sessões de saída para atravessar o NAT, e impede a recepção de pacotes de entrada excepto quando estes fazem parte de uma sessão iniciada a partir de dentro da rede privada, ou seja quando são uma "resposta"[3].

Uma sessão é criada pelo NAT quando um pacote o atravessa, no sentido de dentro para fora (ou seja, do lado privado para o lado público), é identificada pelo quarteto (IP local, porto local, IP remoto, porto remoto) e tem uma duração[3].

O NAT *outbound* rejeita os pacotes quando ambos os nós localizados em redes diferentes, cada um atrás do seu NAT, desejam comunicar entre si, pois quando um nó envia uma mensagem, o NAT do outro nó ignora-a. A ideia base para atravessar um NAT é fazer com que as sessões pareçam ser de saída [3].

2.1 Conjunto de Técnicas

Há várias técnicas em uso para ultrapassar o NAT para que duas máquinas comuniquem entre si, onde quer que estejam (rede pública ou redes privadas). Entre essas técnicas encontram-se: o *Relaying*, *Hole Punching* com UDP do NAT e *Hole Punching* com TCP do NAT.

2.1.1 *Relaying*

Uma dessas técnicas é o *Relaying* [3]. Nesta técnica aplica-se uma abordagem cliente/servidor para fazer *relaying*. Um exemplo, de como esta técnica pode ser aplicada, é termos dois clientes *A* e *B* que iniciaram conexões TCP ou UDP com um servidor *S* cujo endereço é global e tem, por exemplo, o valor de 18.181.0.31:1234. Conforme mostrado na Ilustração 2, os clientes residem em diferentes redes privadas, e os seus respectivos NATs evitam qualquer cliente de iniciar directamente uma ligação para o outro. Em vez de tentar uma ligação directa, os dois clientes podem, simplesmente, usar o servidor *S* para retransmitir as

mensagens entre eles. Por exemplo, para enviar uma mensagem para o cliente *B*, o cliente *A*, simplesmente, envia a mensagem ao servidor *S* através da sua ligação cliente/servidor previamente estabelecida com *S*, e *S* reencaminha a mensagem para o cliente *B* usando a sua ligação cliente/servidor com *B* [3].

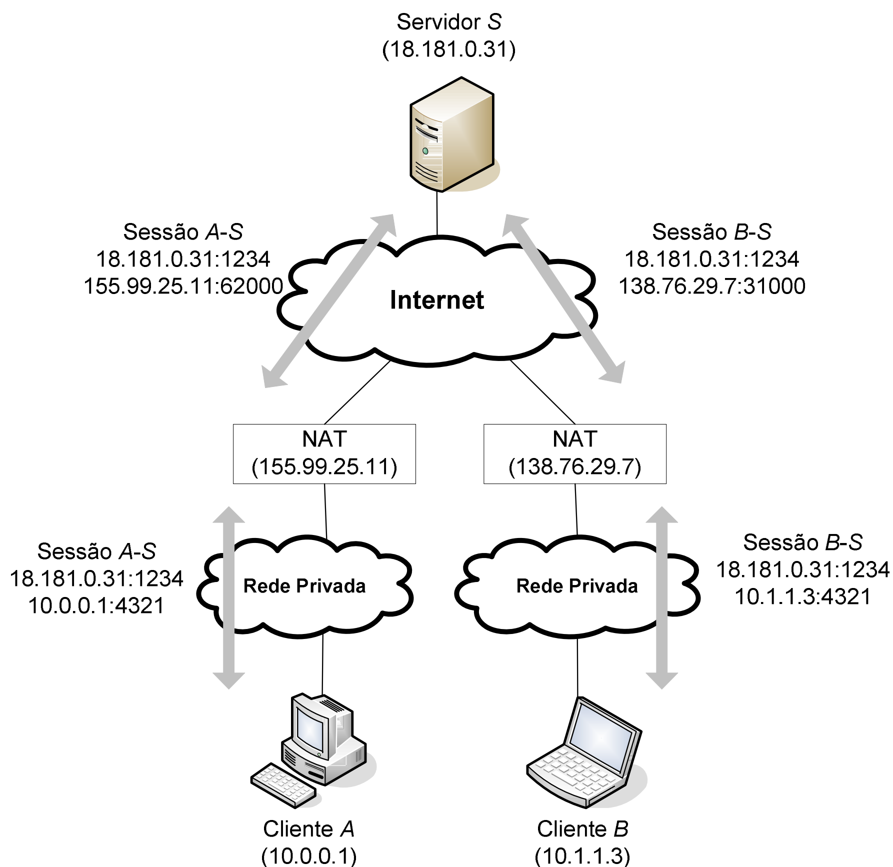


Ilustração 2 - Atravessamento do NAT por *Relaying*

2.1.2 *Hole Punching* com UDP do NAT

Outra técnica, conhecida por *Hole Punching* com UDP [3], usa um servidor de encontro (*rendezvous server*) cujo endereço é global. Nesta técnica os dois clientes *A* e *B* têm sessões UDP activas para o servidor *rendezvous S*. Quando um cliente se regista em *S*, o servidor regista dois *endpoints* para esse cliente: o par (endereço IP, porto UDP) que o cliente acredita estar a usar para falar com *S*, e o par (endereço IP, porto UDP) observados pelo servidor do cliente quando o cliente fala com o servidor. Estes pares têm, respectivamente, a designação de *endpoint* privado e *endpoint* público. O servidor pode obter o *endpoint* privado do cliente a

partir do próprio cliente, num campo localizado no corpo da mensagem de registo do cliente, e obter o *endpoint* público do cliente a partir do endereço IP/Porto de origem indicados no cabeçalho da mensagem de registo UDP. Se o cliente não estiver atrás de um NAT então os *endpoints* privado e público serão iguais.

O cliente *A*, inicialmente, não sabe como atingir *B*, e, portanto, *A* pede a *S* ajuda para estabelecer uma sessão UDP com *B*.

S responde a *A* com uma mensagem contendo os *endpoints* público e privado de *B*. Ao mesmo tempo, *S* usa a sua sessão UDP com *B* para enviar a *B* um pedido de ligação e, nessa mensagem, é enviada os *endpoints* público e privado de *A*. Assim que essas mensagens são recebidas, *A* e *B* conhecem os *endpoints* público e privado um do outro.

Quando *A* recebe os *endpoints* público e privado de *B* através de *S*, *A* começa a enviar pacotes UDP para estes dois *endpoints*, e, posteriormente, "fixa-se" no primeiro *endpoint* que trouxer uma resposta válida a partir de *B*. Da mesma forma, quando *B* recebe os *endpoints* público e privado de *A* no pedido de ligação encaminhado, B começa a enviar pacotes UDP para os *endpoints* de *A* "fixando-se" no primeiro *endpoint* que trouxer uma resposta válida.

Um exemplo concreto de *Hole Punching* com UDP é termos os clientes *A* e *B* com endereços IP privados por trás de NATs diferentes, como apresentado na Ilustração 3. *A* e *B* têm cada um sessões de comunicação UDP dos seus portos locais 4321 para o IP 18.181.0.31:1234 do servidor *S*. No tratamento destas sessões de saída, o NAT *A* atribuiu o porto 62000 no seu próprio endereço IP público, 155.99.25.11, para a utilização de uma sessão com *S*, e o NAT *B* atribuiu porto 31000 para o seu endereço IP, 138.76.29.7, na sessão de *B* com *S*.

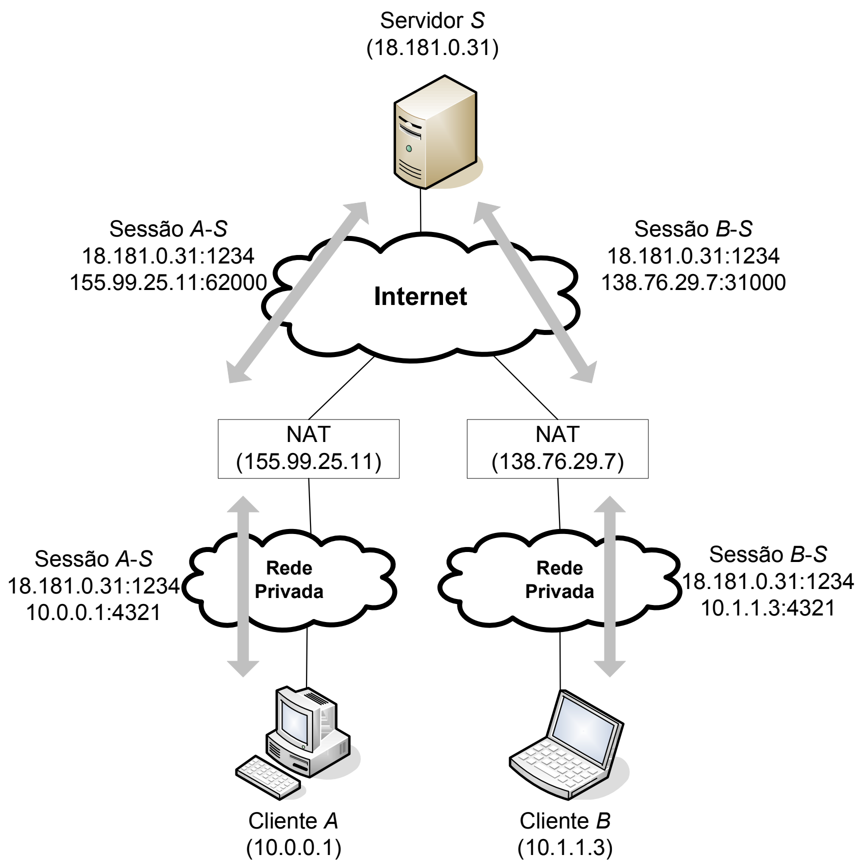


Ilustração 3 - Antes do *Hole Punching* com UDP

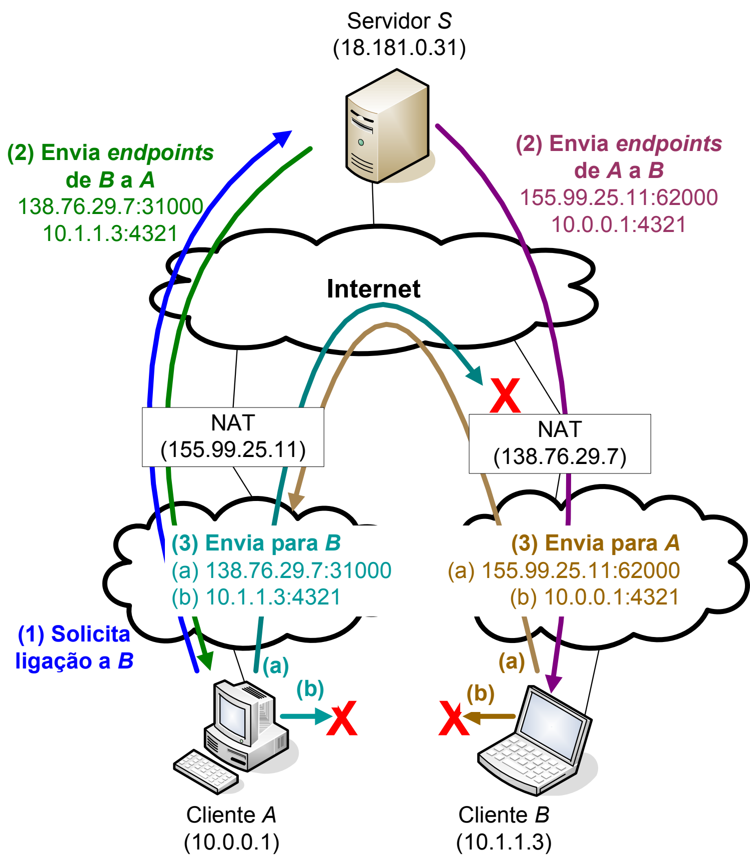


Ilustração 4 - *Hole Punching* com UDP propriamente dito

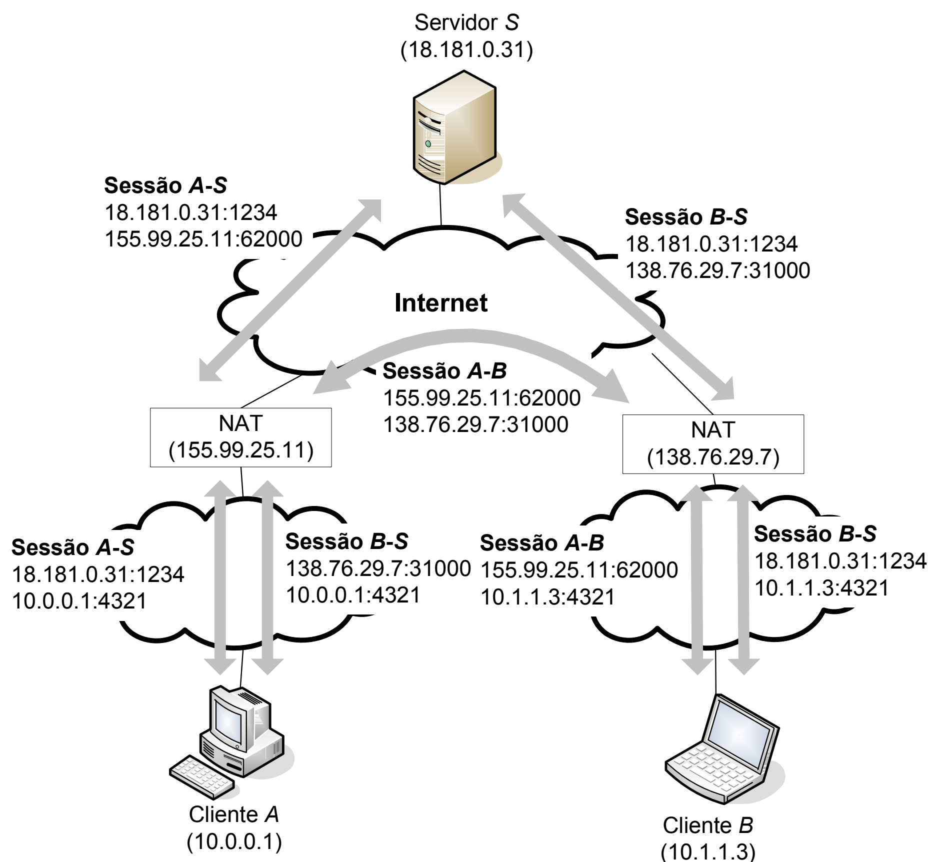


Ilustração 5 - Após *Hole Punching* com UDP

No registo de *A* em *S*, o cliente *A* indica o seu *endpoint* privado (endereço da rede privada) ao *S* como 10.0.0.1:4321. *S* memoriza o *endpoint* privado e público de *A*, sendo este observado por *S* através do endereço indicado no cabeçalho do pacote UDP. O *endpoint* público de *A* é neste caso o 155.99.25.11:62000, sendo este o *endpoint* temporário associado à sessão pelo NAT. Da mesma forma, quando o cliente *B* se regista em *S*, o servidor *S* fica com o seu *endpoint* privado de *B* como 10.1.1.3:4321 e o seu *endpoint* público de *B* como 138.76.29.7:31000.

Depois *A* envia uma mensagem a *S* para este ajudar no estabelecimento de uma ligação com *B*. Em resposta, *S* envia os *endpoints* público e privado de *B* para *A*, e envia os *endpoints* público e privado de *A* para *B*. De seguida, *A* e *B* começam, cada um, a tentar enviar pacotes UDP directamente a cada um desses *endpoints*.

Uma vez que *A* e *B* estão em diferentes redes privadas e seus respectivos endereços IP privados não são globais (*routable*), as mensagens enviadas para os *endpoints* privados atingem o nó errado ou não atingem nenhum nó. As aplicações devem, portanto, autenticar todas as mensagens de alguma forma a filtrar o tráfego “de sonda”. As mensagens podem

incluir nomes específicos da aplicação, *tokens* criptográficos, ou um número aleatório único pré-determinado através de *S*. No entanto se *A* e *B* estivessem na mesma rede privada a troca de mensagens entre os *endpoints* privados seria bem sucedida e a ligação seria estabelecida.

A primeira mensagem de *A* que é enviada para o *endpoint* público de *B*, conforme indicado na Ilustração 4, abre uma sessão de saída no NAT de *A*. Isto porque a mensagem de saída ao passar pelo NAT, "avisa-o" que este é o primeiro pacote UDP numa nova sessão de saída. O *endpoint* da sessão origem (10.0.0.1:4321) é a mesmo que o da sessão existente entre *A* e *S*, mas o seu *endpoint* destino é diferente. Se o NAT *A* comportar-se de forma típica, preserva a identidade *A* do *endpoint* privado, traduzindo todas as sessões de saída a partir *endpoint* origem privada 10.0.0.1:4321 para o *endpoint* correspondente de origem pública 155.99.25.11:62000. *A* primeiro envia uma mensagem de saída para o *endpoint* público de *B*, "fazendo um furo" no NAT de *A* para uma sessão UDP identificada pela extremidade (10.0.0.1:4321, 138.76.29.7:31000) na rede privada de *A*, e através dos *endpoints* (155.99.25.11:62000, 138.76.29.7:31000) na Internet principal.

Se a mensagem de *A* para o *endpoint* público de *B* atinge o NAT de *B* antes da primeira mensagem de *B* para *A* cruzar o próprio NAT de *B*, então o NAT de *B* pode interpretar a mensagem de entrada como o tráfego de entrada não solicitado e ignora-o. A primeira mensagem de *B* para o endereço público de *A*, abre, no entanto, um "buraco" no NAT de *B*, para uma sessão UDP identificada pelos *endpoints* (10.1.1.3:4321, 155.99.25.11:62000) na rede privada de *B*, e os pontos de extremidade (138.76.29.7:31000, 155.99.25.11:62000) na Internet. Uma vez que a primeira mensagem de *A* e *B* tenham atravessado os seus respectivos NATs, os buracos ficam abertos em cada sentido e, dessa forma, a comunicação UDP pode prosseguir normalmente. Depois dos clientes terem verificado que os *endpoints* públicos funcionam, eles podem parar de enviar mensagens para os *endpoints* privados alternativos.

2.1.3 *Hole Punching* com TCP do NAT

Outra técnica de travessia do NAT é a *Hole Punching* com TCP [3]. Em termos de protocolo é semelhante à perfuração UDP mas tem algumas diferenças, pois o TCP assenta na API dos *sockets* Berkeley que foram concebidos em torno do paradigma cliente/servidor, segundo o qual a iniciação de um *socket* entra no modo de "escuta" ou no modo de "envio" (na qual há ligação a um receptor) mas não em ambos simultaneamente.

No entanto, para a perfuração TCP ter sucesso, é preciso utilizar um único porto TCP local para escutar as ligações TCP de entrada de tráfego e, simultaneamente, dar início a múltiplas conexões TCP de saída. Para que isso seja possível é necessário alterar as configurações por omissão dos *sockets*, nas quais se dá indicação para se reutilizar o mesmo endereço e porto, para que no mesmo *socket* seja possível dar inicio a uma ligação em modo de "escuta" e em modo de "envio".

Ao contrário do UDP, onde cada cliente só precisa de um *socket* para comunicar com servidor *rendezvous* e com qualquer número de nós, com o TCP cada aplicação cliente deve gerir vários *sockets* ligados a uma única porto TCP local no nó do cliente, como mostrado na Ilustração 6. Cada cliente precisa de um *socket* de *stream* que representa a sua conexão com o servidor *rendezvous*, um *socket* de escuta para aceitar ligações de entrada do nó "externo", e dois *sockets* de fluxo adicional com o qual deseja iniciar as ligações de saída para os *endpoints* TCP público e privado do nó "externo".

A título de exemplo, considere-se o cenário mais comum no qual os clientes A e B estão atrás de NATs diferentes, como mostrado na Ilustração 3, Ilustração 4 e Ilustração 5, e assuma-se que os números de portos mostrados nas figuras são para o TCP e não UDP. As tentativas de ligação de saída que A e B fazem a cada um dos outros *endpoints* privados falham ou conectam-se ao nó errado. Tal como acontece com o UDP, é importante que as aplicações TCP autentiquem as suas sessões *peer-to-peer*, para evitar o risco de engano de se ligarem ao nó errado na rede local que tem o mesmo endereço IP privado que o nó desejado numa rede privada remota.

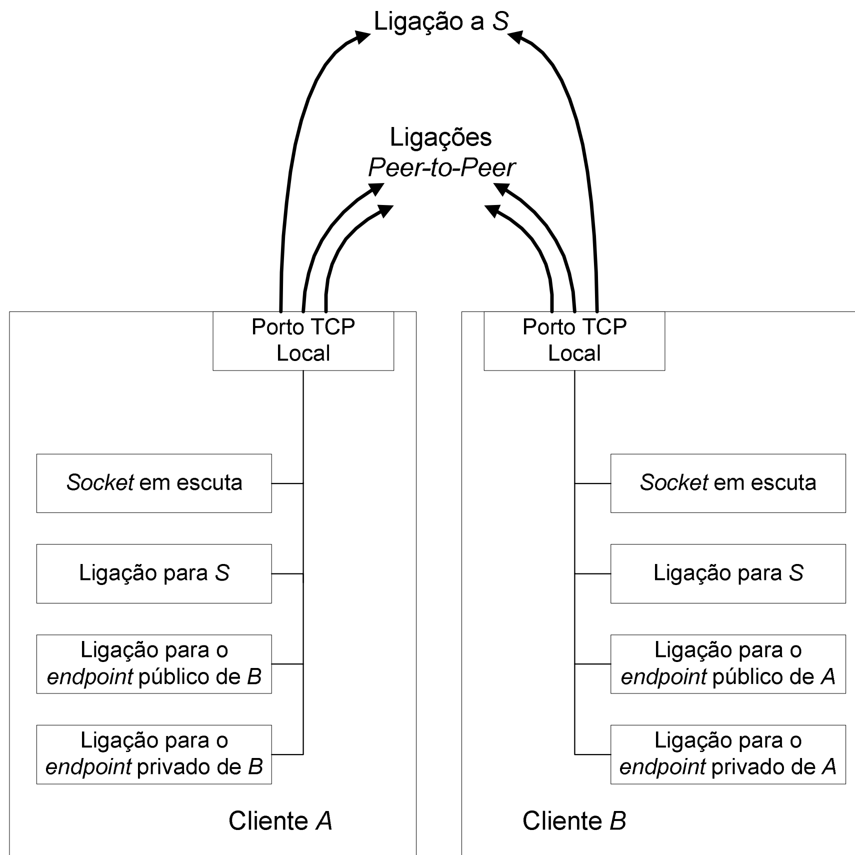


Ilustração 6 - *Sockets* e portos necessários em cada nó para o *Hole Punching* por TCP

As tentativas do cliente de ligação de saída para cada um dos outros *endpoints* públicos, no entanto, fazem com que os NATs respectivos abram novos "buracos" que permitem a comunicação TCP directa entre *A* e *B*. Se os NATs têm um comportamento típico, então uma nova *stream* TCP *peer-to-peer* é formada entre eles. Se o pacote SYN (pacote com *flag* para sincronizar a sequencia de números para iniciar uma conexão [5]) de *A* para *B* atinge primeiro o NAT *B* antes do primeiro pacote SYN de *B* para A atingir o NAT *B*, por exemplo, então o NAT de *B* pode interpretar SYN *A* como uma tentativa de ligação de entrada não solicitada e ignora-o. O primeiro pacote SYN de *B* para *A* deve passar, contudo, porque o NAT de *A* vê este SYN como sendo parte da sessão de saída para *B*, que o primeiro SYN de *A* já havia iniciado.

As técnicas de *Relaying* e de *Hole Punching* têm em comum o facto de precisarem de um servidor através do qual conseguem contactar com o destinatário. No entanto no *Relaying* o servidor serve de “ponte” para a comunicação. No caso do *Hole Punching* o servidor serve de ponto de encontro com o objectivo de sinalizar o destinatário e receber os endereços público e privado do mesmo. Depois, nesta técnica, realiza-se um processamento no sentido de estabelecer a comunicação directamente com o destinatário.

2.2 Software

No mercado há várias aplicações que precisam de estabelecer comunicação com máquinas localizadas em redes distintas, umas na rede pública e outras nas redes privadas. Uma forma encontrada para que a ligação se estabeleça independentemente de onde as máquinas se encontrem é preparar as aplicações no sentido de aplicarem algumas das técnicas mencionadas no capítulo anterior. Entre essas aplicações encontram-se o projecto JXTA, o Skype e o MSN Messenger.

2.2.1 JXTA

O JXTA é um projecto de *open-source* que define um conjunto de protocolos *peer-to-peer*, concebido pela Sun Microsystems, Inc. com a participação de especialistas de instituições académicas e da indústria [6, 7]. Os protocolos JXTA estabelecem uma rede virtual em cima da Internet e de redes não-IP, permitindo que seus *peers* interajam e se organizem, independentemente do seu local de rede (existam ou não *firewalls* e NATs) [6].

Os protocolos JXTA foram concebidos para serem utilizados em qualquer dispositivo de rede, incluindo sensores, telemóveis, PDAs, portáteis, electrodomésticos, routers, computadores desktop, servidores centrais de dados e sistemas de armazenamento, permitindo que comuniquem entre si colaborando mutuamente como entidades computacionais [6, 8].

O projecto da rede JXTA utiliza dois mecanismos primários para *routing* e retransmissão de mensagens. Primeiro, as mensagens JXTA contêm informação de *rounting* como parte dos seus *payloads* úteis. Cada vez que uma mensagem passa por um *hop*, a informação de *payload* é actualizada com as informações *hop* actual. Quando um *peer* recebe uma mensagem, ele pode usar a mensagem de encaminhamento de informações como uma sugestão para o encaminhamento de respostas ao remetente. Em segundo lugar, o projecto JXTA utiliza *peers* especiais chamados *peers* de *relay* para manter informações de *routing*. Qualquer *peer* pode tornar-se num *peer* de *relay*. Os *peers* de *relay* mantêm tabelas de *routing* para retransmitir mensagens para o seu destino. Tanto a informação da mensagem como os *peers* de *relay* são usados para determinar o próximo *hop* para enviar uma mensagem. No JXTA, a técnica de

Relaying é usada para guardar e encaminhar mensagens entre nós que não têm conectividade directa por causa de *firewalls* ou NATs [6, 8].

2.2.2 Skype

O Skype é uma aplicação, que fornece um conjunto de serviços, entre os quais estão os seguintes serviços gratuitos: VoIP (Voz sobre IP) que permite dois utilizadores estabelecerem uma comunicação de áudio; Mensagens Instantâneas que possibilita dois ou mais utilizadores trocarem pequenas mensagens de texto em tempo real; Transferência de ficheiros e Chamadas de vídeo [9-11]. O Skype fornece, também, serviços pagos que permitem aos utilizadores iniciarem e receberem chamadas através de números de telefone regulares através de *gateways* VoIP-PSTN (PSTN = *Public Switched Telephone Network*) [11].

A rede Skype é baseada em super-nós e, dessa forma, organizam os intervenientes em duas camadas: os super-nós e os nós comuns. Normalmente, os super-nós mantêm uma rede entre eles, enquanto os nós ordinários escolhem um, ou um pequeno número de super-nós a que se associam; os super-nós, também, funcionam como nós normais e são eleitos entre eles com base nalguns critérios (como, por exemplo, CPU disponível e IP público). Os nós ordinários fazem pedidos através dos super-nós a que estão associados [10, 11]. Qualquer nó com um endereço IP público, com suficiente CPU, memória e largura de banda da rede é um candidato a tornar-se num super nó. Um *host* ordinário deve-se ligar a um super-nó e deve-se registar no servidor de login do Skype para um login bem-sucedido. Embora não seja um nó próprio Skype, o servidor de login é uma entidade importante na rede Skype. Os nomes de utilizador e senhas são armazenados no servidor de login. A autenticação do utilizador no login, também, é feita no servidor. Este servidor, também, garante que os nomes de login do Skype são únicos em todo o espaço de nome Skype. A Ilustração 7 mostra o relacionamento entre os hospedeiros comuns, e nós super servidor de login. O servidor de login é o único servidor central na rede Skype [10].

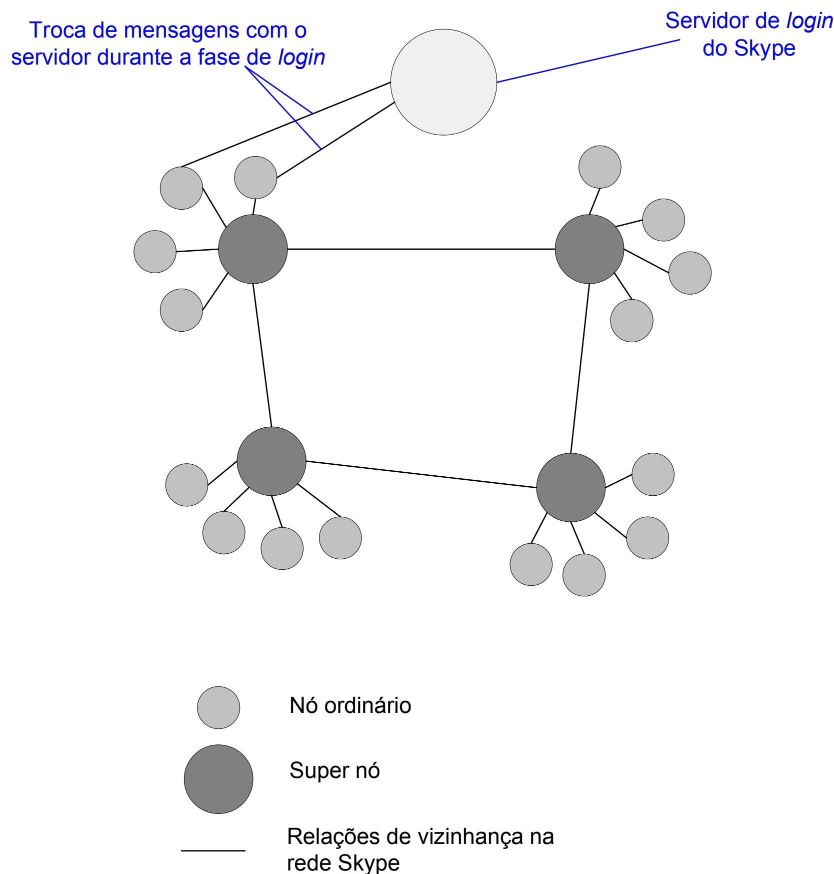


Ilustração 7 - Relacionamento entre os hospedeiros comuns, os super-nós e o servidor de login.

Os nós ordinários enviam controlos de tráfego, incluindo informações de disponibilidade, mensagens instantâneas e pedidos de VoIP e sessões de transferência de ficheiros sobre a rede *peer-to-peer* de super-nós. Se o VoIP ou os pedidos de transferência de ficheiros são aceites, os clientes do Skype estabelecem uma ligação directa entre si. Se os clientes estão atrás de NATs, o Skype usa a travessia NAT do tipo *Hole Punching* para estabelecer a ligação directa [12]. No caso em que a conexão directa falha, o Skype aplica uma abordagem do tipo TURN (*Traversal Using Relay NAT*) onde a sessão é retransmitida por um super-nó publicamente acessível, ou seja, aplica uma técnica de *Relaying* [11]. Esta última abordagem é utilizada quando a travessia do NAT falha, ou uma *firewall* bloqueia certos pacotes do Skype. Assim, o mecanismo global que o Skype utiliza para servir VoIP e pedidos de transferência de ficheiros é bastante robusto, mesmo quando estão presentes as barreiras impostas por NATs e *firewalls*.

2.2.3 MSN Messenger

O programa "MSN Messenger" que é executado no PC liga-se a um servidor da "rede MSN Messenger" através da Internet [13]. Esse programa, é designado por cliente, e envia e recebe

informação para e de outros clientes, via servidor segundo a técnica de *Relaying*. O servidor ao receber a informação, que o cliente lhe envia, processa-a e/ou transmite-a a outros clientes. No caso de um cliente enviar uma mensagem instantânea, faz com que o servidor apenas transfira a mensagem para o destinatário, sendo depois a mensagem processada pelo cliente final.

Numa primeira fase, quando se activa o cliente MSN Messenger estabelece-se uma ligação com um servidor de notificação. Este servidor tem por objectivo lidar com a informação de presença acerca do utilizador e dos utilizadores cuja presença está subscrita. O servidor de notificação realiza outros serviços como notificação ao utilizador de novos *emails* na sua caixa de correio de Hotmail. Permite, também, criar ou juntar-se a sessões de *switchboard*, ou seja, sessões que permitem a troca de mensagens instantâneas entre os clientes. Por outras palavras, cada pessoa num chat MSN corresponde a uma ligação para uma sessão partilhada de *swtichboard*. O *switchboard* actua como um *proxy* entre o utilizador e aquele com quem estiver a falar.

Em termos cronológicos, o primeiro passo a dar quando se realiza uma sessão do MSN Messenger é fazer o registo num servidor de notificação. Se o cliente já tiver um endereço IP guardado de um servidor de notificação, poderá ligar-se directamente a ele. Caso contrário, o cliente deve conectar-se a um servidor *dispath* (DS) localizado num endereço público bem conhecido. O DS é, no fundo, um servidor de notificação por omissão. Se o servidor de notificação ao qual o utilizador está ligado estiver em sobrecarga, será sugerido um servidor com maior disponibilidade para efectuar o "login". Desta forma agiliza-se o processo de autenticação e apresenta-se uma forma de escalar os pedidos, mesmo com a técnica de *Relaying* em uso.
Quando o cliente pretende iniciar uma sessão de conversa ou juntar-se a uma já existente, estabelece uma ligação TCP com um servidor *switchboard* cujo endereço é determinado pelo servidor de notificação. Depois, segundo a técnica de *Relaying* os clientes comunicam entre si. Há, no entanto, comandos de controlo que os clientes enviam para o servidor e que não são reencaminhados para os outros clientes [13].

3 Descrição da Solução

A solução projectada teve em consideração o objectivo de estabelecimento de ligação entre quaisquer nós estejam em redes privadas ou pública e tenham endereços globais ou locais (devido à existência de NAT).

3.1 Arquitectura da Solução

O sistema de comunicações que se pretendeu desenvolver tinha como um dos objectivos, permitir que máquinas localizadas em redes distintas, públicas ou privadas, pudessem interagir. Esta condição determinou o tipo de sistema de computadores a usar que foi um sistema distribuído em detrimento de um sistema centralizado.

Era, também, necessário que houvesse garantia de entrega das mensagens enviadas pelos nós, assim como assegurar que a ordem pela qual eram enviadas era respeitada. Esta exigência determinou o protocolo de transporte a ser usado, que foi o TCP em vez do UDP. O TCP fornece um serviço fiável ponto a ponto, pois possui mecanismo de controlo de erros, retransmissão implícita e controlo de fluxo, contrariamente ao UDP que não tem controlo de erros nem garante a sequência [5]. Assim, a técnica de *Hole Punching* com UDP para ultrapassar o NAT foi posta de parte.

A travessia por *Hole Punching* com UDP ou TCP não funciona com todos os NATs [2, 3]. Num estudo feito por Bryan Ford, Pyda Srisuresh e Dan Kegel concluiu-se que 82% dos NATs eram compatíveis com a técnica de *Hole Punching* com UDP e que somente 64% dos NATs permitiam o *Hole Punching* com TCP [3]. O método de *Relaying* por sua vez funciona sempre, desde que ambos os clientes se consigam ligar ao servidor [3]. Assim, e como o trabalho pretendia garantir a conectividade entre os nós, optou-se por utilizar a técnica de *Relaying*. Esta técnica assenta no paradigma cliente/servidor e apresenta algumas desvantagens face às técnicas de *Hole Punching* “geradoras” de ligações *peer-to-peer*. Entre essas desvantagens estão o consumo de processamento do servidor, a ocupação maior de largura de banda e o aumento da latência da comunicação entre os nós [3]. No entanto, o *Relaying* constitui a técnica mais fiável para atravessar um NAT [3] e como se pretendia

maximizar o sucesso da conectividade optou-se por esta técnica. A Ilustração 8 mostra a arquitectura adoptada baseada na técnica de *Relaying*.

Conforme indicado, pretendia-se que a ordem das mensagens enviadas fosse respeitada na entrega. Isto é garantido pelo protocolo de transporte apenas entre os dois nós que estabeleceram a ligação TCP. Como se vai optar pela técnica de *Relaying*, há duas ligações TCP entre cada dois clientes (ligação cliente emissor/servidor e servidor/cliente receptor) e, por isso, é necessário que o projecto implemente um mecanismo que garanta a ordem de envio. Caso contrário a informação ao passar pelo *relay* podia sofrer alteração na sua ordem.

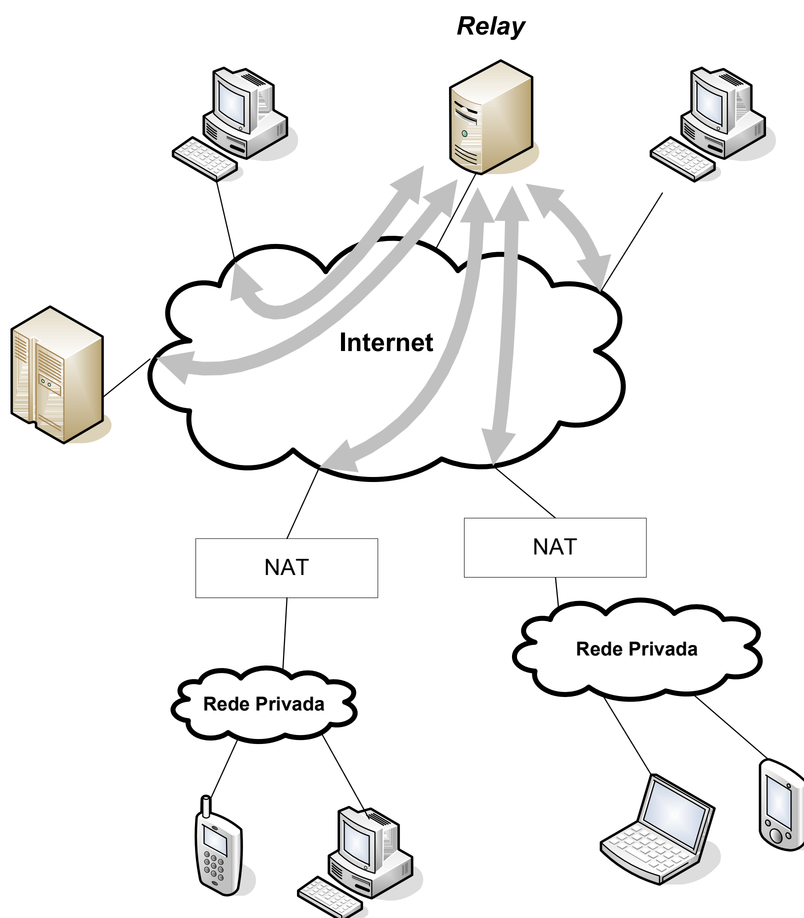


Ilustração 8 - Arquitectura adoptada para as comunicações

O sistema de comunicações, para além de ter que lidar com as barreiras criadas por NATs, tem, também, que fornecer um conjunto de serviços que proporcione a troca de mensagens entre grupos de utilizadores.

A localização de nós por parte de um cliente pressupõe a existência de estado, onde fica registado o endereço do respectivo participante. Se esse estado estiver centralizado a busca e

tratamento dessa informação é facilitada, pois lançar a busca sobre todo o sistema de elementos e fazer o tratamento de informação espalhada pode resultar num processamento moroso e complexo. A máquina de *Relay* está acessível por todos os intervenientes e, por isso, optou-se por usá-la para guardar o estado sobre os elementos que partilham a plataforma de comunicações. Assim, a Ilustração 9 evidencia a arquitectura física da solução.

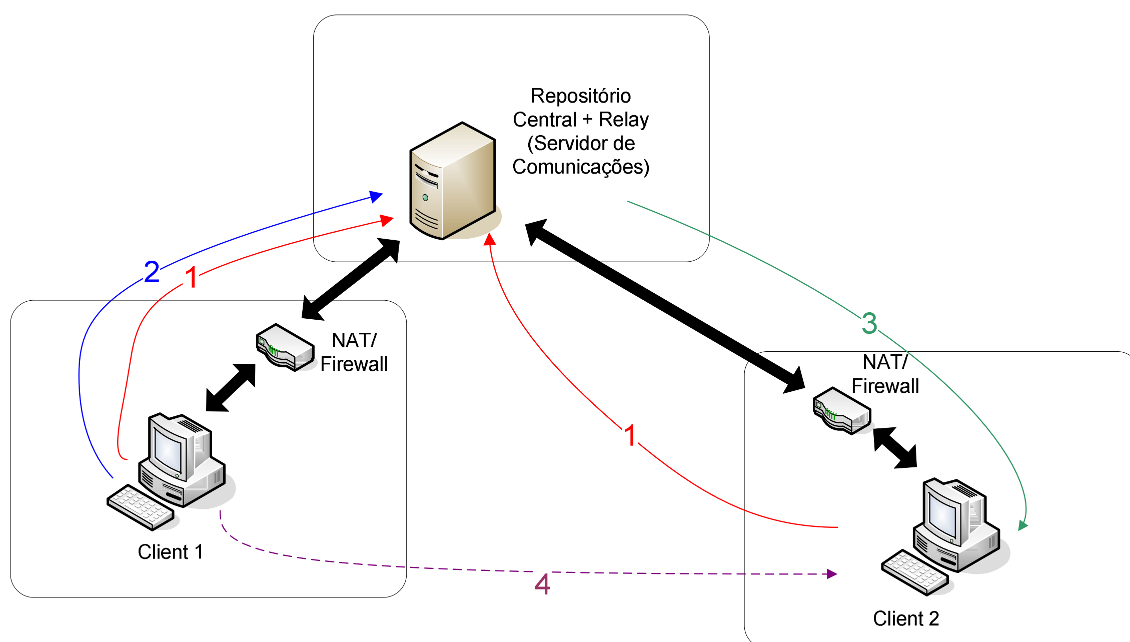


Ilustração 9 - Arquitectura Física da Solução

De acordo com esta arquitectura qualquer cliente faz, numa primeira fase, uma ligação TCP ao *relay* (setas 1). Desta forma, qualquer cliente cria uma sessão no NAT quando o primeiro pacote o atravessa, no sentido de dentro para fora (ou seja, do lado privado para o lado público). Depois, quando o cliente 1 quiser comunicar com o cliente 2, envia uma mensagem para o *Relay* (seta 2), que por sua vez reencaminha a mensagem para o cliente 2 (seta 3). O fluxo da seta 3 é sempre possível porque houve inicialmente o fluxo da seta 1 que criou a sessão NAT. Se o cliente 1 quisesse enviar directamente uma mensagem para o cliente 2 poderia não conseguir por causa de barreiras adicionais, tais como, um NAT que existisse entre eles sem uma sessão iniciada impedindo a comunicação.

Nesta arquitectura todos os clientes têm que se registar no Repositório Central. O Repositório Central serve como ponto central de registo e de *relay*. Um determinado cliente quando pretende contactar outro tem de o fazer via *Relay*. Como o cliente final recebe mensagens do cliente inicial através do *Relay* não há problemas com o NAT porque o canal foi previamente

aberto no sentido cliente/*Relay*. Com esta arquitectura apenas se exige que haja um endereço IP público (o do *relay*).

3.2 Comunicações de Grupos

A troca de mensagens entre elementos cooperantes necessita que a entrega de mensagens *multicast* seja garantida. Ora o domínio "intra" *multicast* (dentro de uma LAN) está quase sempre disponível (é necessário, no entanto, que o *router* aceite tráfego *multicast*) mas num domínio "inter" *multicast* poderão surgir dificuldades na sua implementação. Uma dessas dificuldades é criada por alguns ISPs que se mostram relutantes em fornecer uma área ampla de serviço de *routing multicast* [14]. Assim, e para garantir o *multicast*, implementou-se uma base de apoio às comunicações de grupos. Um grupo é, portanto, um conjunto dos processos cooperantes, reunidos segundo um determinado critério, definido pelo sistema ou pelo utilizador, e que pode ser endereçado como uma unidade única [15].

A Ilustração 9 evidencia a forma como dois nós, numa arquitectura cliente/servidor, trocam mensagens. O servidor é a máquina que desempenha funções de *relay* e à qual todos os clientes estão ligados. Os nós podem cooperar entre si na realização de uma tarefa e, por isso, formar um grupo. Caso os nós pertençam ao mesmo grupo, fica facilitada a tarefa de endereçamento de informação àquele conjunto de PCs. Os serviços de comunicações de grupos visam facilitar a troca de mensagens na forma de um para muitos ou de muitos para muitos [16].

O grupo por sua vez é um conjunto de processos distribuídos que cooperam na realização de uma tarefa. Na solução apresentada para a plataforma de comunicações, optou-se pela existência do grupo "Main" ao qual todos os elementos pertencem. Assim, o endereçamento de um *broadcast* consiste em enviar uma mensagem ao grupo "Main".

A aplicação consumidora da plataforma de comunicações pode, portanto, contar com o grupo "Main" e com os grupos que entretanto criou. Assim, à medida que os utilizadores se vão registando no sistema, vão sendo adicionados ao grupo "Main" através do coordenador de grupos do servidor. Depois se, por exemplo, os clientes 1, 2 e 6 estiverem a realizar uma tarefa em conjunto, os clientes 3 e 4 estiverem a cooperar noutra operação e o cliente 5 não estiver associado a nenhuma tarefa de grupo, teremos vários grupos conforme a Ilustração 10.

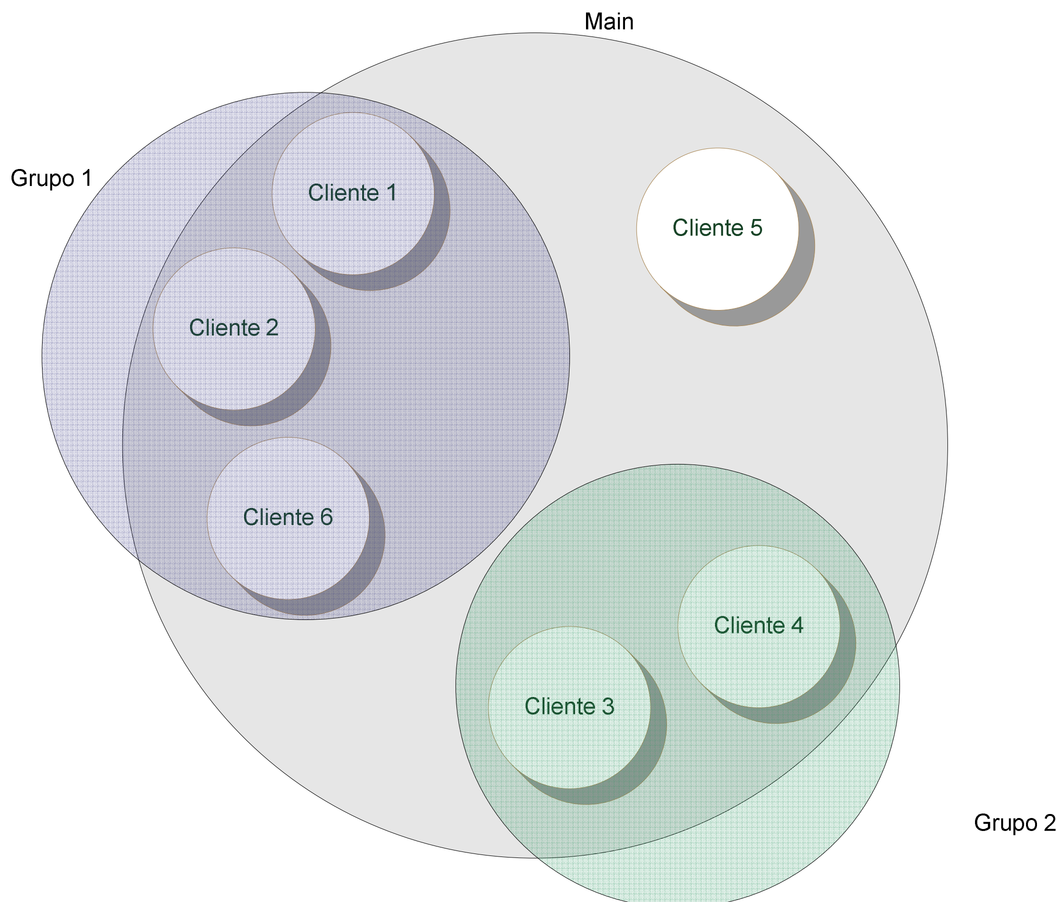


Ilustração 10 - Exemplo de uma disposição de grupos conforme as necessidades da aplicação num determinado instante

Um determinado cliente pertence sempre ao grupo "Main" e pode pertencer a outros grupos. Um determinado grupo contém sempre pelo menos um elemento excepto o grupo "Main" que pode estar vazio. O registo do novo nó num determinado grupo gera eventos nos elementos desses grupos a avisar a chegada de um novo colaborador. O novo elemento, também, receberá alguma informação de contexto, nomeadamente, dados referentes ao controlo de mensagens de sequência FIFO.

3.3 Módulos Lógicos da Solução

A solução implementada é composta por módulos de software que estão reflectidos na Ilustração 11.

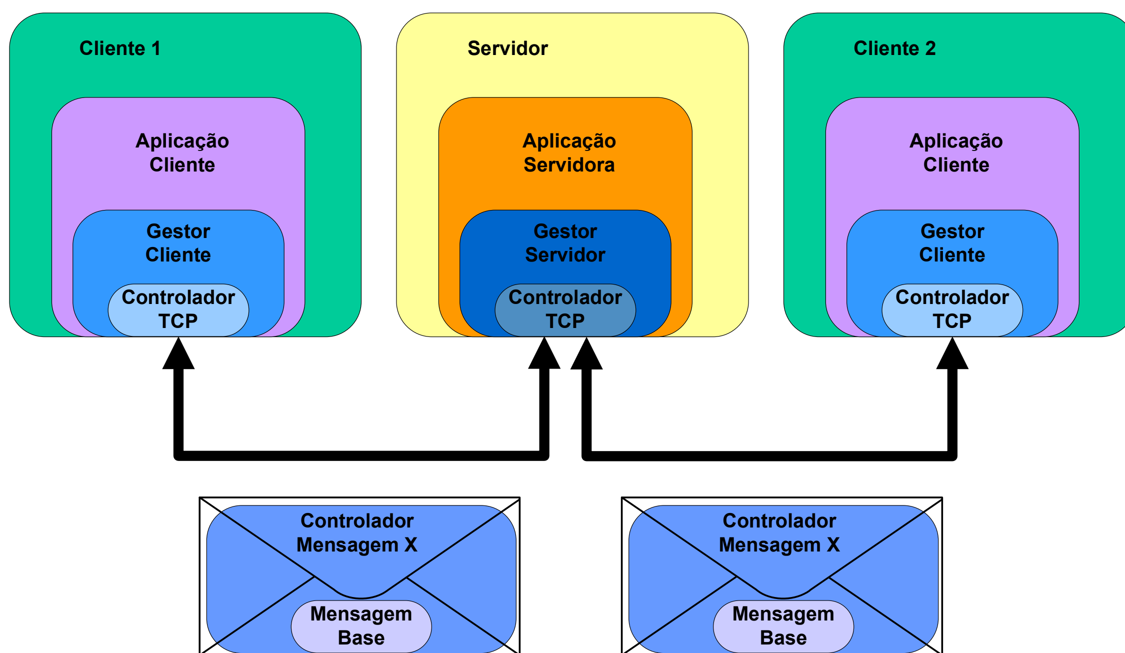


Ilustração 11 - Relação entre o software da aplicação consumidora e da plataforma de comunicações

Os módulos de software foram criados tendo em consideração os serviços que prestam à aplicação consumidora da plataforma de comunicações. Esses serviços estão classificados em três tipos:

- Serviços de iniciação e finalização;
- Serviços de notificação (ocorrem em determinados eventos);
- Serviços que podem ser solicitados pela aplicação.

Os serviços de iniciação e finalização estão relacionados com o arranque e paragem do servidor e clientes TCP. Os serviços de notificação ocorrem de forma automática quando acontecem determinados eventos (quando um cliente termina a sua ligação, todos os clientes são notificados dessa ocorrência). Os serviços que podem ser solicitados pela aplicação são aqueles que são desencadeados pela aplicação consumidora.

Para facilitar as mensagens *multicast*, alguns desses serviços permitem que os destinatários sejam uma lista de clientes, podendo estes formar um grupo.

Vejamos agora os diferentes módulos que compõem a plataforma de comunicações.

No cliente, temos, dentro da aplicação o "Gestor do Cliente". Este módulo de software é responsável pelo tratamento de comunicações no lado do cliente, sendo composto por código e estado. O código gere as funcionalidades base das comunicações e dos serviços de iniciação (ligação ao servidor) e finalização (terminar a ligação ao servidor). Para além disso, este código presta apoio à concretização de serviços que podem ser invocados pela aplicação cliente. É o módulo "Gestor do Cliente" que possui as estruturas de dados necessárias à realização dos serviços. O cliente possui ainda o módulo "Controlador TCP" (ver Ilustração 11) localizado dentro do "Gestor de Cliente". Este módulo controla a ligação TCP estabelecida com o servidor, sendo, portanto, responsável pelo transporte de dados para o servidor e recepção de dados provenientes dele. Quando envia dados, estes são originários do módulo "Gestor de Cliente" e, após serem seriados, são entregues ao módulo "Controlador TCP" do servidor. Quando o "Controlador TCP" recebe dados, entrega-os ao "Gestor" que os reconstrói.

O servidor possui uma estrutura semelhante à do cliente. O "Gestor do Servidor" possui estruturas de dados necessárias às funcionalidades base e aos serviços de iniciação (arranque do servidor) e finalização (paragem do servidor). Tal como no cliente, existe o "Controlador TCP" que é responsável por enviar dados ao cliente (provenientes do "Gestor do Servidor") e receber dados do cliente. O envio e recepção dos dados, exige a seriação e a reconstrução respectivamente, dos dados.

As mensagens trocadas entre os intervenientes (Cliente→Servidor e Servidor→Cliente) têm como objectivo contribuir para a realização de um serviço que requer transporte (de acordo com o protocolo definido). Para facilitar o tratamento da mensagem criou-se o módulo "Controlador de Mensagem" que possui o código que invoca as funcionalidades do gestor destinatário do interveniente corrente ("Gestor do Servidor" ou "Gestor do Cliente") de forma a contribuir para a concretização do serviço. O "Controlador de Mensagem" é diferente para cada serviço e possui um conjunto de campos necessários à realização do serviço em causa. A mensagem trocada entre os intervenientes corresponde sempre à seriação de um "Controlador de Mensagem". Vejamos um exemplo:
Se o cliente 1 solicitar o serviço de criação do grupo "Grupo1", o respectivo "Controlador de Mensagem" terá um campo com o nome do grupo a criar, cujo conteúdo é "Grupo1". Este controlador possui também o código a executar no "Servidor". O controlador é seriado e transportado para o Servidor. Assim, ao chegar ao servidor, a mensagem é reconstruída para

"formar" novamente o "Controlador da Mensagem". Através do código que possui, o controlador solicita ao "Gestor do Servidor" que verifique se esse grupo já existe e, se não existir, pede-lhe para criá-lo. Ainda através desse código, o controlador regista a resposta (se criou ou não o grupo) e pede que a mensagem seja transportada para o remetente (para este poder consultar a resposta).
Vejamos outro exemplo. Se o cliente 1 quiser solicitar o serviço de envio de dados de aplicação ao cliente 2, é utilizado o "Controlador de Mensagem" apropriado. Vamos supor que os dados a enviar são um ficheiro e o respectivo nome. Neste caso, estamos a "entrar" no protocolo da aplicação. O cliente 1 deverá começar, por exemplo, por criar um objecto correspondente à instância de uma classe "serializável" da aplicação, com dois campos: "NomeFicheiro" que é uma *string* e "ConteudoFicheiro" que é um *array* de bytes. Depois do objecto estar, devidamente, preenchido (com o nome e conteúdo do ficheiro) será posto no campo de dados do "Controlador de Mensagem" (este campo aceita qualquer tipo de dados desde que sejam "serializáveis"). O controlador é, então, seriado e enviado para o servidor. Ao chegar ao servidor, este controlador é instanciado após ser reconstruído. Este controlador, através do código que possui solicita ao "Gestor do Servidor" que seja transportado para o seu destinatário final (cliente 2). Assim, o gestor seria o controlador e transporta-o para o cliente 2. Ao chegar ao cliente 2, o controlador é reconstruído e, através do código que possui, pede ao "Gestor do Cliente" que avise o cliente final de que chegaram dados. O "Gestor do Cliente" informa, então, o cliente de que chegou dados. Como o controlador não tem mais código para executar o serviço termina. O cliente final faz agora o que quiser com os dados mas, em princípio, vai interpretá-los e "perceber" que se trata de um ficheiro.

O "Controlador da Mensagem" possui ainda a "Mensagem Base". A "Mensagem Base" possui o tipo de informação comum a todos os controladores. Entre essa informação estão os "endereços" do remetente e destinatários e a lista de respostas dadas durante a execução do serviço. Os "endereços" referidos são considerados pelos gestores (do cliente e do servidor) quando realizam a função de transporte do "Controlador da Mensagem". A "Mensagem Base" possui também código que permite, por exemplo, extrair uma resposta específica de um determinado destinatário.

3.4 Serviços Disponibilizados pela Plataforma de Comunicações

A plataforma de comunicações disponibiliza um conjunto de serviços a utilizar pela aplicação consumidora. Esta aplicação distribuída terá sempre um ou mais servidores e vários clientes. Um dos servidores da aplicação irá alojar a parte servidora da plataforma de comunicações e os clientes irão conter a parte cliente do sistema de comunicações.

Os serviços disponibilizados pela plataforma de comunicações podem ser agrupados, conforme visto no ponto 3.3, da seguinte forma:

- Serviços de iniciação e finalização;
- Serviços de notificação (ocorrem em determinados eventos);
- Serviços que podem ser solicitados pela aplicação.

Vejamos agora, esses serviços com mais detalhe.

3.4.1 Serviços de iniciação e finalização

A iniciação do servidor de comunicações é feita através do método **StartTCPServer**. Esta acção é desencadeada pelo servidor da aplicação consumidora, tipicamente, quando arranca. Para este serviço são passados diversos parâmetros, entre os quais estão o IP e o Porto do servidor de comunicações. Este serviço para além de activar um servidor de comunicações TCP cria também o grupo "Main" ao qual vão pertencer todos os clientes. A gestão das comunicações TCP é, totalmente, gerida pela plataforma de comunicações e, por isso, a aplicação consumidora, tanto na vertente servidora como cliente, é-lhe alheia.

O serviço de iniciação do cliente – **Start** é executado quando a aplicação consumidora da plataforma de comunicações pretende registar-se no sistema de comunicações (gerido pelo servidor). Esta acção ocorre, tipicamente, no arranque da aplicação cliente. O método que implementa este serviço recebe, como parâmetros, o IP e a Porto do servidor de comunicações. O registo do cliente no servidor de comunicações implica também o seu registo no grupo "Main". Uma excepção será despoletada durante este serviço se o registo não for bem sucedido (por exemplo, se o servidor de comunicações estiver em baixo).

Nenhum outro serviço poderá ser executado pelo cliente enquanto ele não estiver registado no sistema.

Os serviços de iniciação executam o código associado à preparação de todas as estruturas de dados que lhes dão suporte (a iniciação do servidor de comunicações, por exemplo, inicia a *HashTable* de grupos com o grupo "Main").

O serviço de finalização do cliente - **Stop** deverá ser executado quando a aplicação consumidora do sistema de comunicações não precise de consumir mais serviços de comunicação. A execução deste serviço acontece, normalmente, quando a aplicação cliente está a terminar. O método que implementa este serviço não contém parâmetros. O "desregisto" do cliente no servidor tem, como consequência, o "desregisto" do cliente no grupo "Main".

A finalização do servidor de comunicações é feita segundo o método **StopTCPServer**. A chamada a este método ocorre através do servidor da aplicação consumidora. Esta acção ocorre, tipicamente, quando se faz "shutdown" ao servidor da aplicação.

A Tabela 3 e Tabela 4 do Anexo F mostram os protótipos dos métodos dos serviços mencionados atrás.

3.4.2 Serviços de notificação (ocorrem em determinados eventos)

Alguns dos serviços prestados pela plataforma de comunicações são realizados de forma automática quando ocorrem determinados eventos (alteração da composição de um grupo e recepção de dados).

Um desses serviços é o **NewElement** que é prestado sempre que um determinado grupo passa a ter um novo membro. Isso acontece, por exemplo, quando o cliente se regista na plataforma de comunicações e, consequentemente, é adicionado ao grupo "Main". Esta acção faz desencadear eventos em todos os membros já existentes do grupo "Main" no sentido de serem avisados de que um novo elemento passou a fazer parte do grupo.
Complementarmente, a plataforma, envia para o novo colaborador o estado referente ao controlo das mensagens FIFO que possam existir para o grupo em questão, visto que as

ligações TCP garantem a sequência da informação apenas entre dois nós. Mas, no caso de um cliente enviar um conjunto de mensagens a outro cliente, a ordem com que elas chegam ao cliente final não é garantida pelo TCP, porque as mensagens são enviadas via servidor Relay e, aqui, a ordem, pode ser alterada. Para que a ordem seja mantida, as mensagens precisam de ser marcadas com os atributos referentes ao tratamento FIFO.

Outro desses serviços é o **QuitElement** e ocorre quando um membro sai de um determinado grupo. Neste caso, todos os membros que ainda pertencerem a esse grupo são informados dessa ocorrência.

Na categoria de serviços de notificação existem também os serviços **DataArrivedFromEmitter** e o **DataArrived**. Estes serviços são despoletados como consequência da solicitação do serviço **Request_SendData** (ver ponto 3.4.3) que transporta dados (que não precisam de resposta) de um cliente para um conjunto de clientes destinatários. Esses dados ao passarem pelo servidor de comunicações e antes de serem enviados aos destinatários levam à execução do serviço de notificação **DataArrivedFromEmitter**, que avisa o servidor de que estão a passar dados. Se a aplicação servidora subscrever o evento deste serviço então pode efectuar um processamento a esses dados (pode, por exemplo, ter um contador geral de mensagens que não precisam de resposta). Após os dados serem entregues a cada destinatário, este é notificado pelo serviço **DataArrived** de que chegaram dados para si.

Há ainda os serviços de notificação **DataWithAnswerArrivedFromEmitter**, **DataWithAnswerArrived** e o **DataWithAnswerArrivedToEmitter**. Estes serviços são accionados como resultado da execução do serviço **Request_SendDataAndReceiveAnswer** (ver ponto 3.4.3), que transporta dados (com os pedidos) de um cliente para um conjunto de clientes destinatários e, depois, carrega as respostas desses clientes para o emissor. Os dados, durante o transporte, ao passarem pelo servidor, no sentido emissor/destinatários accionam o serviço de notificação **DataWithAnswerArrivedFromEmitter** (no servidor). Quando os dados chegam aos destinatários é prestado o serviço "**DataWithAnswerArrived**" (no cliente) e, no *handler* do evento deste serviço, é suposto o cliente definir a sua resposta (se o cliente receptor não definir uma resposta o emissor aguarda até ocorrer um *timeout*). Depois, as respostas são transferidas para o cliente emissor e, ao passarem no servidor (sentido destinatários/emissor) accionam o serviço **DataWithAnswerArrivedToEmitter**. Os serviços de notificação que ocorrem no servidor (**DataWithAnswerArrivedFromEmitter** e

DataWithAnswerArrivedToEmitter) podem ser subscritos e, nos respectivos *handlers* dos eventos associados pode-se, por exemplo, ter código para calcular o tempo médio que o servidor demora a satisfazer um pedido com resposta.

Para se usufruir dos serviços de notificação a aplicação cliente terá de subscrever os eventos referentes aos serviços pretendidos.

A Tabela 5 do Anexo G indica os eventos e respectivos *delegates* accionados pelos serviços.

3.4.3 Serviços que podem ser solicitados pela aplicação

Existe um conjunto de serviços que, após o registo no sistema de comunicações, o cliente pode solicitar.

Um desses serviços é o **Request_WhoAmI** que interroga o servidor de comunicações quanto ao ID atribuído ao cliente solicitante.

Outro desses serviços é o **Request_SendData** que transporta dados (aviso) de um cliente emissor para um conjunto de destinatários (identificados segundo uma lista de IDs ou um nome de grupo). A concretização deste serviço começa com o transporte dos dados para o servidor. Após chegar ao servidor, e, caso tenha indicado um grupo como destinatário (em vez de uma lista de IDs de clientes receptores), o grupo é descodificado para uma lista de IDs. Depois é prestado o serviço de notificação **DataArrivedFromEmitter** (ver ponto 3.4.2). De seguida os dados são transportados para os clientes destinatários. Após cada receptor de dados receber os dados é prestado o serviço de notificação **DataArrived** (ver ponto 3.4.2).

O serviço **Request_SendDataAndReceiveAnswer** transporta dados (pedido) de um cliente para uma lista de clientes destinatários, aguarda as respostas destes e depois carrega as respostas para o cliente emissor. Este serviço inicia-se com o transporte da mensagem para o servidor. Depois o nome do grupo, caso seja indicado, é descodificado para uma lista de clientes. De seguida é prestado o serviço de notificação **DataWithAnswerArrivedFromEmitter** (no servidor, ver ponto 3.4.2). Depois os dados são encaminhados para os destinatários. Em cada destinatário, a recepção de dados, leva à execução do serviço de notificação **DataWithAnswerArrived** (ver ponto 3.4.2). O receptor,

que supostamente subscreveu o evento do serviço **DataWithAnswerArrived**, deverá, no respectivo *handler*, atribuir uma resposta. Depois a resposta é enviada ao servidor. Este por sua vez colecciona as respostas dos vários destinatários. Após recepcionar todas as respostas (ou ocorrer *timeout*) é executado o serviço de notificação **DataWithAnswerArrivedToEmitter** (no servidor, ver ponto 3.4.2). O serviço termina com a entrega das respostas ao emissor.

Outro serviço que pode ser solicitado pelo cliente é o **Request_CreateGroup**. Através deste serviço, o cliente pode criar novos grupos de membros.

Existe também o serviço **Request_AddMeToGroup** que permite que o cliente faça parte de um grupo já existente.

A plataforma disponibiliza ainda o serviço **Request_RemoveMeFromGroup** que permite que o solicitador saia de um determinado grupo.

O serviço **Request_GetGroupsNames** permite obter um *array* com todos os nomes dos grupos existentes.

Outro serviço que pode ser solicitado pelo cliente é o **Request_GetGroupClientIds** que devolve o conjunto de colaboradores que fazem parte de um determinado grupo.

Há também o serviço **Request_GetLocalEndPoints** permite obter os endereços locais de todos os clientes registados.

Por fim, tem-se o serviço **Request_GetClientEndPoint** que permite obter o endereço com que um determinado cliente está registado no servidor de comunicações (endereço resultante do NAT se este existir).

A Tabela 6 do Anexo H apresenta as assinaturas e uma descrição dos métodos usados nos serviços que os clientes podem solicitar.

O servidor de comunicações para além dos serviços que presta de forma automática tem um que pode ser solicitado através do servidor da aplicação. Esse serviço chama-se **Request_HelloWorld** e permite enviar uma mensagem a um determinado cliente.

A Tabela 7 do Anexo H indica qual a assinatura do serviço **Request_HelloWorld**.

3.5 Codificação de um serviço

A realização de um serviço da plataforma de comunicações usa uma mensagem assíncrona (sem retorno de dados) ou síncrona (com retorno de dados). A mensagem assíncrona pode ser usada para enviar avisos e, quando é usada, o emissor não fica bloqueado à espera de um resultado (a mensagem é enviada e o emissor prossegue com o seu código). A mensagem síncrona deverá ser usada num contexto em que o emissor pretende fazer um pedido e aguarda uma resposta como retorno. Neste caso o emissor fica bloqueado, não prosseguindo com o código até receber a resposta (ou dar *timeout*).

A mensagem é endereçada ao servidor de comunicações que pode ou não reencaminhá-la para um conjunto de destinatários. Se a mensagem tiver retorno de dados então o servidor aguarda pelas respostas dos destinatários e entrega-as ao requerente do serviço.
A codificação de um serviço depende do tipo de mensagem que usa (síncrona/assíncrona) e, se for passível de ser invocado pela aplicação, concretiza-se com o desenvolvimento de um método a inserir no Gestor (do Cliente ou Servidor, conforme o serviço pretendido) e uma classe que implementa a interface "IMsgCtrl" - Controlador da Mensagem (ver Ilustração 11).

3.5.1 Natureza das Mensagens usadas pelos serviços

Vejamos agora com maior detalhe os tipos de mensagens usadas nos serviços.
As mensagens usadas pelos serviços são de dois tipos:

- As que funcionam como avisos, de semântica assíncrona;
- As que operam como pedidos (têm informações de retorno), do semântica síncrona.

As mensagens do tipo aviso são apenas reencaminhadas para os seus destinatários, não se aguardando eventuais respostas.

Os pedidos, no entanto, requerem um conjunto de estruturas de dados que dão suporte ao protocolo de comunicações. Não basta que um determinado elemento do grupo escreva o pedido na *stream* TCP de output do *socket* e, de seguida, leia a resposta na *stream* de input. O motivo prende-se com o facto de na *stream* de input poder estar a chegar outra resposta ou um pedido de outro cliente.

De acordo com o que foi definido na arquitectura da solução Ilustração 9, verifica-se que a mensagem antes de atingir o cliente final, passa pelo *relay*, que por sua vez reencaminha a mensagem para os destinatários. Se a mensagem tiver informação de retorno, então o *relay* colecciona as respostas dos vários destinatários e entrega-as ao emissor.

O método responsável pela concretização de um pedido, começa por incrementar o contador geral de pedidos. O objectivo é identificar cada pedido de forma inequívoca. De seguida regista na tabela de pedidos informações sobre o pedido corrente e inicia um controlador de tempo para forçar um *timeout* em pedidos que demorem muito a satisfazer. Todo este fluxo ocorre no processo cliente que faz o pedido (emissor). De seguida, o controlador inicia a contagem de tempo enquanto a mensagem é transportada para o servidor de comunicações. O servidor processa o pedido e devolve a resposta ao emissor. Se a resposta demorar a vir do servidor e entretanto ocorrer um *timeout* (por omissão é 10 segundos podendo ser configurável) o fluxo prossegue sem a resposta do servidor. Depois, o temporizador de tempo é parado e de seguida a entrada do pedido registada na tabela de pedidos é eliminada. A entrada na tabela é identificada pelo ID do pedido que consta na resposta. Por fim devolve-se a resposta ao método que efectuou o pedido. No caso de ter ocorrido *timeout* a resposta é nula (sem resultados).

Há, no entanto, possibilidade de distinguir uma resposta nula motivada por um *timeout* de uma resposta "null" dada por um destinatário. Para explicar isso segue-se o seguinte exemplo. O cliente 1 enviou um pedido ao conjunto C no qual constam os clientes 2, 3 e 4. O cliente 2 respondeu "R", o cliente 3 respondeu *null* e o cliente 4 demorou muito tempo a responder até que ocorreu um *timeout*. O que o cliente 1 recebe é uma lista de respostas no formato *<long,object>* em que *long* é o ID do destinatário e o *object* é a sua resposta. Assim, o cliente 1 recebe {(2,"R"),(3,null)}. Ou seja, não recebe a entrada referente ao cliente 4. Perante o facto de o número de respostas ser incompleto, pois o nº de respostas é diferente do nº de destinatários (os destinatários são preenchidos pelo servidor – que traduziu o nome do grupo num conjunto de IDs) o emissor descarta tudo ou aproveita o que pode.

Os fluxos de pedidos com e sem *timeout* são descritos pelos diagramas da Ilustração 12 e Ilustração 13 respectivamente.

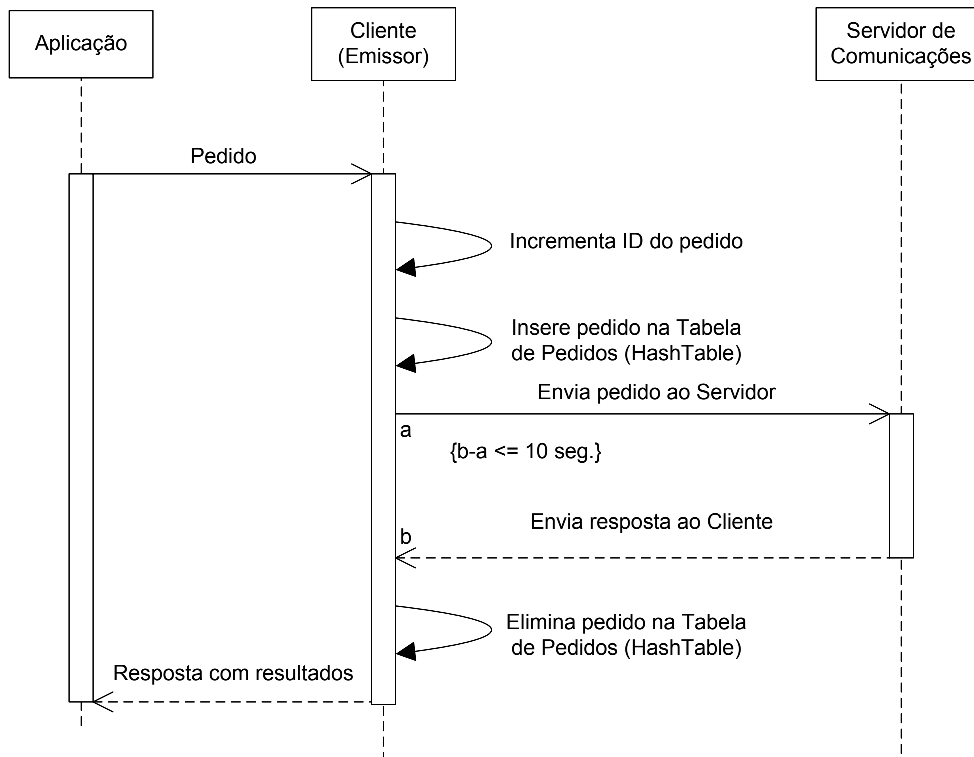


Ilustração 12 - Diagrama de sequência para um pedido com retorno de dados e sem ocorrência de *timeout*

Um exemplo de um pedido com retorno é o procedimento de criação de um grupo. O procedimento inicia-se com o registo na tabela de pedidos do cliente que faz o pedido (emissor). A seguir o controlador da mensagem (ver ponto 3.5.2) do pedido é endereçado ao servidor de comunicações. O servidor (que funciona também como registo central dos clientes) verifica se o nome para o novo grupo é válido e se ele ainda não existe. O servidor responde ao emissor quanto ao sucesso da criação do grupo. Se o servidor demorar muito tempo a responder o fluxo prossegue com uma resposta nula. O emissor recebe a resposta referente ao pedido (True indica que foi criado o grupo; False, não foi criado o grupo; Null ocorreu *timeout*) que tinha registado na tabela de pedidos e elimina esta entrada, libertando os recursos criados para o controlo do pedido.

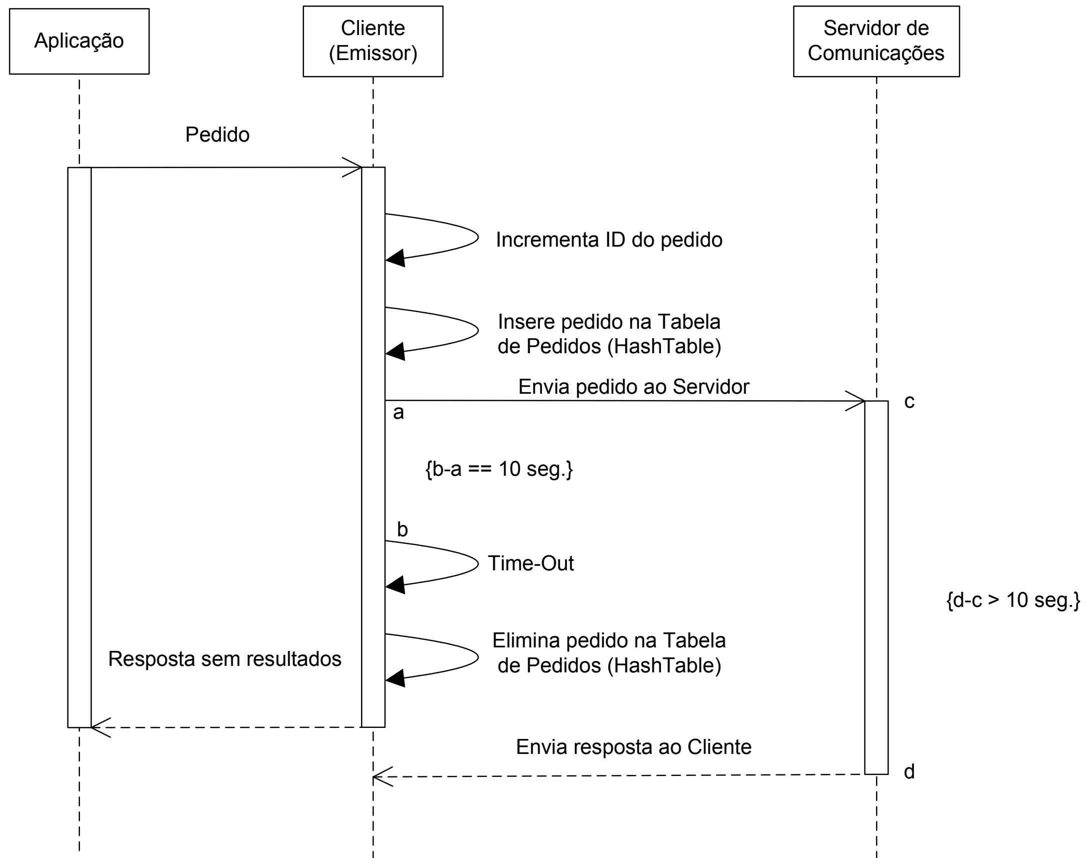


Ilustração 13 - Diagrama de sequência para um pedido com retorno de dados e com ocorrência de *timeout*

3.5.2 Controlador da Mensagem

O controlador da mensagem é uma classe que concretiza a parte principal de serviço que requer transporte de mensagem. Esta classe implementa a interface "IMsgCtrl" que requer a codificação de dois métodos nos quais constam o código a ser executado no servidor (RunOnServer) e o código a ser executado nos clientes (RunOnClient) destinatários.

Os controladores de mensagens estão todos localizados no mesmo *assemblie* que é distribuído com o servidor e os clientes finais. Assim, o código do controlador de execução no servidor (RunOnServer) e no cliente (RunOnClient) já se encontra na posse destes. No entanto, por questões de simplificação de criação de um serviço, os métodos a executar no cliente e no servidor estão centralizados numa classe que implementa a interface "IMsgCtrl" conforme Código 1.

```
public interface IMsgCtrl
    {
        //Propriedade com a "Mensagem Base" que possui a informação
        //comum a todos os controladores. Entre essa informação estão
        //os "endereços" do remetente e destinatários.
        IMsg Msg { get; }

        //Método com o código a executar no servidor de comunicações.
        //clientID = Identificação do cliente que enviou a mensagem (emissor).
        //tcpServerCtrl = Controlador das comnunicações TCP no sevidor. Possui
        //    o contexto de execução (disponibiliza, por exemplo, a lista de
        //    utilizadores activos)
        void RunOnServer(long clientId, ITCPServerCtrl tcpServerCtrl);

        //Método com o código a executar no cliente final (receptor da mensagem).
        //clientId = Identificação do cliente que está a receber a mensagem (receptor).
        //tcpClientCtrl = Controlador das comnunicações TCP no cliente final
        //    (receptor). Possui o contexto de execução (disponibiliza, por
        //    exemplo, a lista FIFO de mensagens a entregar).
        void RunOnClient(long clientId, ITCPClientCtrl tcpClientCtrl);
    }
```

Código 1 - Interface do Controlador de Mensagem

A interface "IMsgCtrl" possui a propriedade "Msg" (implementa a interface "IMsg") que tem o tipo de informação base da mensagem comum a todos os serviços. Nessa informação consta a identificação do remetente, a lista de destinatários (ou o nome do grupo), o *timeout* com o tempo limite para satisfazer o pedido, a lista de respostas entre outros dados. O objecto referenciado por esta propriedade, possui também vários métodos entre os quais se encontra um que permite os destinatários registarem as suas respostas.

O estado da plataforma de comunicações está distribuído pelo servidor (no Gestor do Servidor) e pelos clientes (no respectivo Gestor do Cliente, ver Ilustração 11). No servidor existe um conjunto de tabelas para tratamento dos grupos e dos pedidos que requerem resposta. No cliente existe também um conjunto de tabelas para tratamento dos pedidos e das mensagens FIFO. As estruturas de dados que compõem o estado são detalhadas no ponto 3.3.

O método "`RunOnServer`" contém o código a executar no servidor de comunicações. Para este método são passados dois parâmetros: o *id* do cliente emissor e a referência para o Gestor do Servidor (possui o estado da plataforma de comunicações do servidor).

O método "`RunOnClient`" será executado em todos os clientes finais a que se destina a mensagem. Para este método são passados os parâmetros: *id* do cliente final (receptor) e a referência para o gestor das comunicações TCP do cliente receptor (possui o estado referente às comunicações do cliente).

Todos os serviços que requerem transporte de mensagem, são implementados com o auxílio de controladores de mensagens. Estes controladores possuem o código a executar no servidor

de comunicações e nos clientes finais. No entanto, a aplicação consumidora da infra-estrutura de comunicações não precisa de implementar novos controladores a não ser que pretenda estender os serviços desta infra-estrutura. Os controladores por omissão prestam os serviços necessários à criação de um protocolo de aplicação.

Um exemplo de concretização de um serviço através de um controlador de mensagem é mostrado no Código 2. Este serviço devolve uma lista de *endpoints* locais dos clientes pretendidos e do servidor de comunicações (caso seja indicado na lista de destinatários) ao solicitador.

```
//Controlador para obter os Endpoints locais dos destinatários
[Serializable]
public class CMsgCtrl_GetLocalEndpoint : IMsgCtrl
{
    //Mensagem a transportar que contem entre outras coisas, a lista
    //de destinatários de que se quer obter os endpoints.
    private IMsg _msg;

    //Constructor para o qual se passa a mensagem.
    public CMsgCtrl_GetLocalEndpoint(IMsg msg)
    {
        _msg = msg;
    }

    //Propriedade que permite obter a mensagem
    public IMsg Msg
    {
        get { return _msg; }
    }

    //Código a executar no servidor de comunicações.
    public void RunOnServer(long clientId, ITCPServerCtrl tcpServerCtrl)
    {
        //Verifica se o servidor de comunicações também é um destinatário de que
        //se pretende obter o endpoint (o ID do servidor é sempre -1).
        //Em caso afirmativo o servidor regista a sua resposta através
        //de SetAnswer
        if (_msg.RequestedIds.Contains(-1))
            _msg.SetAnswer(-1, tcpServerCtrl.GetLocalEndPoint());

        //De seguida o servidor reencaminha (através de relaying) o pedido
        //para a lista de clientes destinatários e espera pelas respostas (*)
        bool wasTimeOut;
        tcpServerCtrl.SendRequest(this, out wasTimeOut);

        //Após receber todas as respostas dos destinatários ou quando ocorrer
        //timeout, o servidor envia a resposta ao cliente emissor que fez o
        //pedido
```

```
            _msg.MsgType = EMsgType.Answer;
            tcpServerCtrl.SendMsgToClient(clientId, this);
        }

        //Código a executar no cliente final (receptor)
        public void RunOnClient(long clientId, ITCPClientCtrl tcpClientCtrl)
        {
            //O cliente final regista a resposta com "SetAnswer"
            _msg.SetAnswer(clientId, tcpClientCtrl.GetLocalEndPoint());
            _msg.MsgType = EMsgType.Answer;

            //O cliente final envia a resposta ao servidor. O servidor está, nesta
            //fase a colectar respostas (está no ponto (*))
            tcpClientCtrl.SendMsgToServer(this);
        }
    }
```

Código 2 - Controlador "GetLocalEndpoint"

A utilização deste controlador exige, através do seu construtor, a indicação da informação base da mensagem a enviar, onde consta, entre outra informação, a lista de destinatários de que se pretende obter os *endpoints* locais.

No método “`RunOnServer`” está codificado um bloco de código que começa por registar o *endpoint* local do servidor (se este for um dos destinatários). Depois o servidor através de *relaying* reenvia para todos os destinatários o pedido do *endpoint* local. Após o reenvio o servidor aguarda que todos os destinatários respondam ou que ocorra um *timeout*. Os destinatários respondem logo após a recepção do controlador com a execução do método “`RunOnClient`”. O método “`RunOnClient`” termina com o envio da resposta para o servidor. O servidor que ainda está a executar o método “`RunOnServer`” vai coleccionando todas as respostas que chegam dos destinatários. Após todas as respostas terem chegado, o servidor termina a execução do método “`RunOnServer`” com o reenvio das respostas para o cliente emissor. O diagrama de sequência visualizado na Ilustração 14 mostra o fluxo de chamadas originado pelo controlador referido.

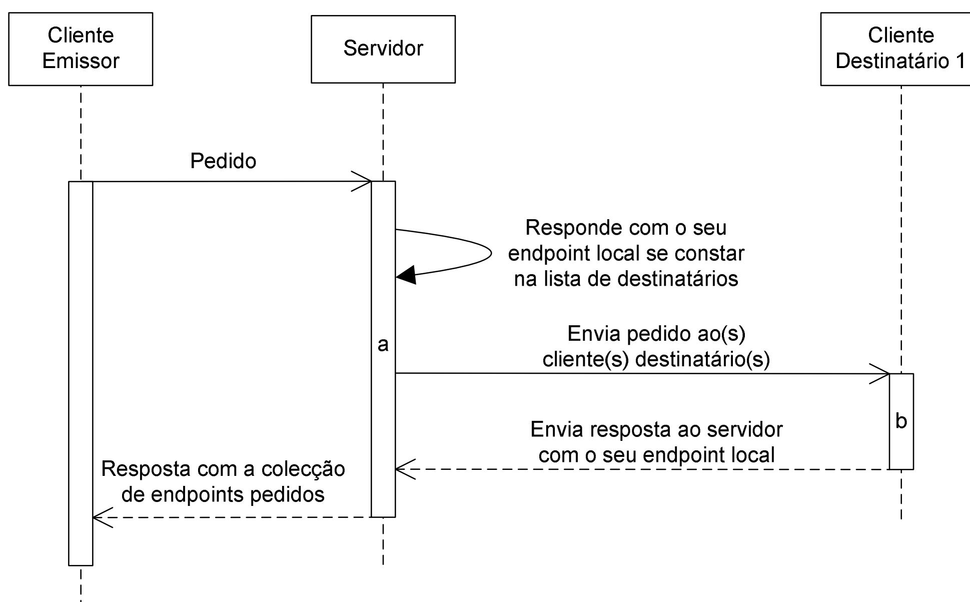


Ilustração 14 - Diagrama de sequência para o controlador CMsgCtrl_GetLocalEndpoint (que devolve a lista dos *endpoints* locais dos destinatários)

Na Ilustração 14 está identificado a execução do método "`RunOnServer`" (a) e a realização do método "`RunOnClient`" (b).

A informação trocada entre os intervenientes do serviço corresponde sempre à seriação do controlador da mensagem em causa e à sua respectiva dimensão em bytes conforme indicado na Ilustração 15. Assim, o emissor serializa o controlador e envia ao receptor.

4 bytes	n bytes
Dimensão	Seriação do Controlador da Mensagem

Ilustração 15 - Formato da informação trocada pelos intervenientes

O receptor, por sua vez ao receber dados na sua *stream* de entrada, determina o seu tamanho (interpretando os primeiros quatro bytes) e lê o conteúdo dos dados propriamente ditos, sobre o qual efectua a "desserialização".

A dimensão é indicada através de um valor de quatro bytes porque os clientes podem trocar informações extensas entre si, nomeadamente ficheiros de grande dimensão. Um ficheiro com

o tamanho de 20 MB tem uma dimensão razoável de ser aceite pelo sistema. Como 20 MB são 20971520 bytes e, para os descrever são necessários 25 bits, ou seja, 4 bytes.

3.5.3 Mensagem Base

A mensagem base faz parte de qualquer controlador de mensagem e, possui entre outra informação, a lista de destinatários.

A mensagem base é uma classe com estado e comportamento para o tratamento base de uma mensagem. A mensagem base implementa a interface "IMsg" e faz parte do controlador de mensagem (ver Código 1). A interface "IMsg" está evidenciada no Código 3.

```
public interface IMsg
{
    long RequesterId { get; }

    long RequestId { get; set; }

    string OrderGroup { get; set; }

    long OrderNumber { get; set; }

    EMsgType MsgType { get; set; }

    List<long> RequestedIds { get; set; }

    string RequestedGroupName { get; }

    List<CAnswer> Answers { get; }

    object GetAnswer(long requestedId);
    bool HasAnswer(long requestedId);

    int TimeOut { get; set; }

    void SetAnswer(long requestedId, object answer);
    void SetAnswers(List<CAnswer> answers);

    bool AlreadyFinished { get; }

    DateTime CreateDateTime { get; }

    bool PutMeOutOfTheGroup { get; set; }
}
```

Código 3 - Interface "IMsg"

A propriedade “RequesterId” permite consultar qual foi o emissor da mensagem.

A propriedade “RequestId” permite guardar ou consultar o nº do pedido atribuído pelo emissor. Um pedido termina com a entrega da resposta ao cliente solicitador. Nesta fase o solicitador consulta o ID do pedido e localiza nas suas estruturas de dados a informação associada ao pedido. Quando já não precisa do ID do pedido, o requerente elimina toda a informação associada ao pedido em causa.

As propriedades “OrderGroup” e o “OrderNumber” são usadas nas mensagens em que se pretende entrega de acordo com a ordem FIFO. A propriedade “OrderGroup” representa um grupo de mensagens virtual ao qual se quer atribuir uma numeração. Enquanto que a propriedade “OrderNumber” indica o nº da mensagem FIFO dentro do “OrderGroup”.

A propriedade “MsgType” indica se a mensagem é um aviso (não tem retorno de dados) é um pedido (tem retorno de dados) ou é uma resposta. De acordo com este tipo de mensagem a plataforma de comunicações ajusta o seu comportamento. Por exemplo, se um determinado cliente recebe uma resposta, consulta a sua estrutura de dados para ligar a resposta ao seu pedido. Dando outro exemplo, se o cliente está a receber um pedido então vai processá-lo no sentido de poder enviar uma resposta.

A propriedade “RequestedIds” é a lista de destinatários (opcionalmente pode indicar-se um nome de um grupo em “RequestedGroupName”).

A propriedade “RequestedGroupName” é o nome do grupo a que se destina a mensagem (pode ser “” se em vez de um grupo se indicou uma lista especifica de destinatários).

A propriedade “Answers” possui as respostas fornecidas à mensagem do tipo “pedido”.

O método “GetAnswer” permite obter a resposta dada por um determinado destinatário.

O método “HasAnswer” permite verificar se existe uma resposta de um determinado destinatário.

A propriedade “TimeOut” permite indicar ou consultar o tempo limite para satisfazer um pedido.

O método “SetAnswer” permite que o destinatário especifique a sua resposta a uma mensagem do tipo “pedido”.

O método “SetAnswers” permite a indicação de várias respostas (usado, por exemplo, pelo servidor quando após colectar um conjunto de respostas, constrói uma mensagem com as mesmas).

A propriedade “AlreadyFinished” indica se o pedido já possui todas as respostas desejadas. Esta propriedade é, por exemplo, consultada pelo servidor e, caso retorne “True” significa que pode enviar as respostas ao cliente emissor.

A propriedade “CreateDateTime” possui a data e hora em que o pedido foi criado.

A propriedade “PutMeOutOfTheGroup” permite que o emissor seja excluído do grupo a que se destina a mensagem.

A Ilustração 16 exemplifica a relação entre as interfaces IMsgCtrl e IMsg e os controladores CMsgCtrl_DataWithAnswer, CMsgCtrl_Data e CMsgCtrl_CreateGroup. Nesta figura verificamos que os controladores de mensagens implementam a interface IMsgCtrl. A propriedade Msg dos controladores tem uma classe que implementa IMsg na qual consta o estado e o comportamento para o tratamento base de uma mensagem. As respostas às mensagens do tipo pedido são uma lista de CAnswer que guarda o ID do destinatário e a respectiva resposta.

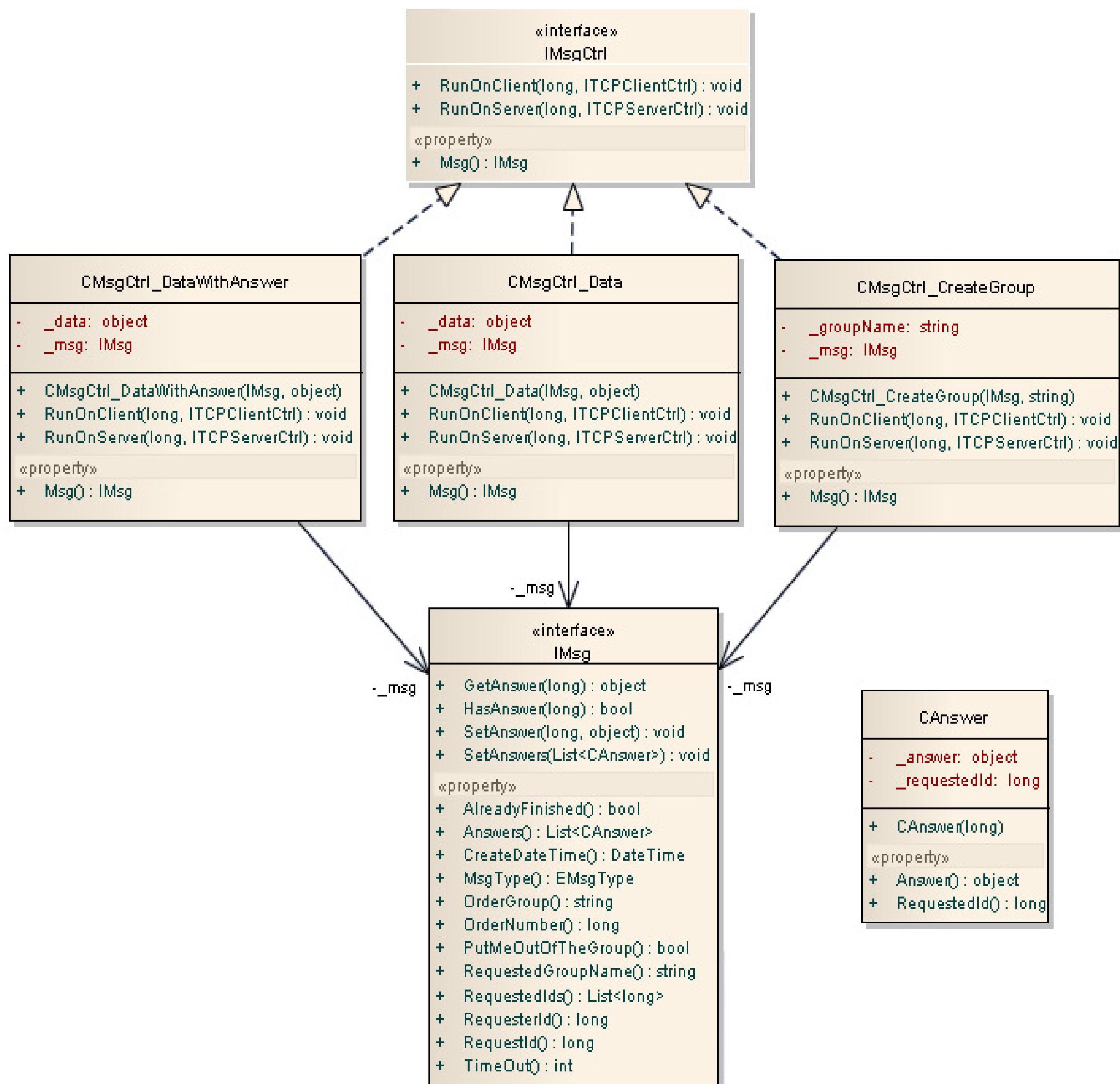


Ilustração 16 - Diagrama UML que evidencia a relação entre as interfaces IMsgCtrl e IMsg e os controladores

3.6 Camadas de Software

Os clientes comunicam entre si via *relay* (servidor) por *sockets* TCP/IP. A camada responsável, nos clientes, pelo tratamento dos dados (resultado da seriação de um controlador de mensagem que implementa IMsgCtrl) enviados e recebidos está materializada na livraria dinâmica XTCPClient. A camada no servidor encarregada da emissão e recepção de dados está localizada fisicamente na livraria dinâmica XTCPServer. A Ilustração 17 pretende esclarecer a forma como estas camadas de software interagem.

O código comum ao cliente e servidor, como por exemplo os controladores de mensagens e as interfaces usadas nos seus contextos, está no ficheiro XTCPShared.dll. Assim, o sistema de comunicação está fisicamente distribuído por três ficheiros: XTCPClient.dll, XTCPServer.dll e XTCPShared.dll.

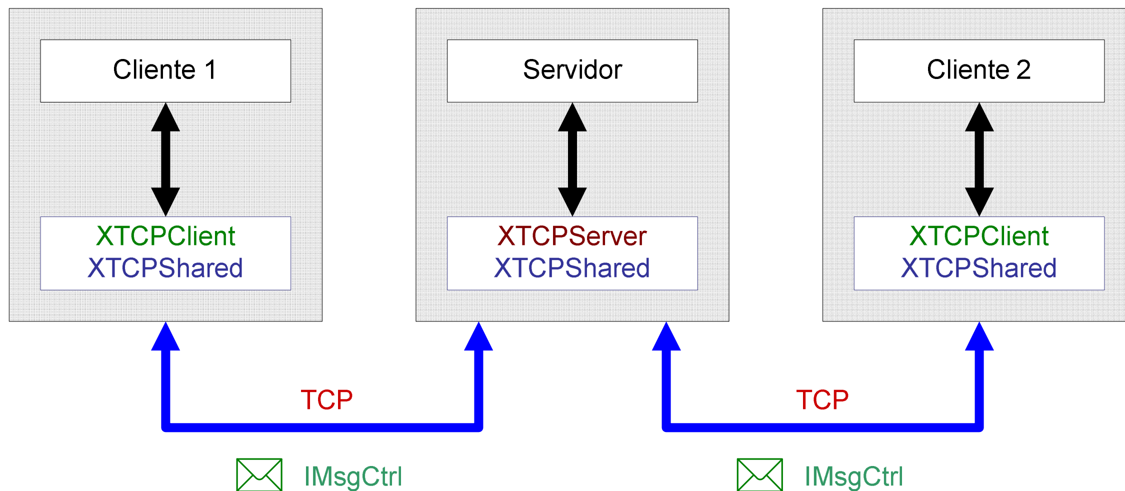


Ilustração 17 - Interacção entre as camadas de software do sistema de comunicações

3.6.1 Camada XTCPClient

A camada XTCPClient possui os módulos de software "Gestor de Cliente" e "Controlador TCP" (ver Ilustração 11) referentes ao cliente. Esta camada é usada pelo cliente da plataforma de comunicações e, através dela o cliente controla o *socket* TCP. Para a aplicação consumidora do sistema de comunicações esta camada virtualiza o acesso às comunicações. Uma das principais classes desta dll é a CTCPClientCtrl que está localizada no "Gestor de Cliente" e cujo conteúdo permite manipular o contexto do sistema de comunicações referente ao cliente assim como aceder aos seus serviços base. Esta classe tem várias estruturas de dados, que suportam os serviços que fornecem, das quais se destacam:

- A tabela de pedidos;
- A tabela de mensagens FIFO recebidas;
- A tabela dos contadores das mensagens FIFO a enviar.

A tabela de pedidos permite ligar as respostas aos pedidos previamente feitos, o que é indispensável para as mensagens com retorno (com resposta).

A estrutura da tabela de pedidos é a indicada na Ilustração 18.

Pedidos (_myRequests)

Chave	Valor	
ID do Pedido (long)	Mensagem (CMsg)	Controlador de Tempo (CRunTimeOut)

Ilustração 18 - Estrutura da Tabela de Pedidos localizada no cliente

Na chave da tabela de pedidos, regista-se o ID do pedido cujo número é dado por um contador sequencial existente no contexto do cliente.

No valor da tabela temos a mensagem (CMsg) e o controlador de tempo.

O controlador do tempo (CRunTimeOut) permite gerar um *timeout* caso o pedido não seja cumprido dentro do tempo máximo estipulado.

A tabela de mensagens recebidas para tratamento FIFO serve para o tratamento de mensagens que requerem ordem (ordem entre o emissor e o receptor para um determinado grupo de comunicações e para um grupo virtual de ordenações). Caso o emissor envie um conjunto de mensagens cuja sequência deva ser respeitada, então os receptores tem que garantir o seu tratamento FIFO. Os clientes receptores têm, através da tabela de mensagens para tratamento FIFO, a informação necessária para contemplar uma mensagem porque está na sua vez de ser tratada ou colocá-la num repositório porque chegou fora de ordem.

A tabela de mensagens FIFO recebida pelo cliente da plataforma de comunicações (através do serviço automático quando o cliente entra num grupo onde existem emissores com envio de mensagens FIFO) tem o seguinte formato indicado na Ilustração 19.

Mensagens recebidas para tratamento FIFO (_receiveFIFOMsgs)

Chave			Valor		
Nome do Grupo (string)	Grupo de Ordenação (string)	ID do Emissor (long)	Nº da próxima mensagem (long)	ChaveB	ValorB
				Nº da mensagem recebida (long)	Mensagem (IMsgCtrl)

Ilustração 19 - Estrutura da Tabela de Mensagens Recebidas para Tratamento FIFO

Nesta tabela guardam-se as mensagens que precisam de tratamento FIFO. A chave compõe-se pelo Nome do Grupo (grupo que identifica um conjunto de clientes), Grupo de Ordenação e identificação do emissor. O Grupo de Ordenação é um grupo virtual que permite haver vários conjuntos de mensagens FIFO para o mesmo grupo de clientes e vindos do mesmo emissor, mantendo assim uma sessão de conversação de mensagens ordenadas oriundas de um determinado emissor. Assim, pode-se ter, por exemplo, o grupo "Jogos" com o emissor 1001 a transmitir a mensagem 56 do grupo de ordenação "Jogo1". Simultaneamente, o mesmo emissor pode estar a transmitir a mensagem 34 do grupo de ordenação "Jogo2". Pode, portanto, haver diferentes contadores para o mesmo grupo e emissor.

No valor tem-se o número da próxima mensagem esperada (que corresponde à próxima mensagem a ser tratada) e outra tabela que funciona como repositório das mensagens recebidas mas cujo número sequencial não era o esperado. Por exemplo, esperava-se a mensagem 56 que ainda não foi recebida mas já se recebeu as mensagens 57 e 58. Estas mensagens (57 e 58) estão guardadas no campo valorB da tabela repositório (os IDs 57 e 58 são guardados na chaveB). Assim que se receber a mensagem 56 (próxima mensagem aguardada) trata-se a mensagem 56, 57 e 58 e limpa-se o repositório. Em relação ao Jogo2 o cliente corrente aguarda pela mensagem 34 embora já tenha recebido a mensagem 37. A Ilustração 20 pretende demonstrar este exemplo.

Mensagens recebidas para tratamento FIFO (_receiveFIFOMsgs)

<table>
<tr><th colspan="3">Chave</th><th colspan="3">Valor</th></tr>
<tr><td rowspan="2">Nome do Grupo (string)</td><td rowspan="2">Grupo de Ordenação (string)</td><td rowspan="2">ID do Emissor (long)</td><td rowspan="2">Nº da próxima mensagem (long)</td><td>ChaveB</td><td>ValorB</td></tr>
<tr><td>Nº da mensagem recebida (long)</td><td>Mensagem (IMsgCtrl)</td></tr>
<tr><td rowspan="3">Jogos</td><td rowspan="3">Jogo1</td><td rowspan="3">1001</td><td rowspan="3">56</td><td>ChaveB</td><td>ValorB</td></tr>
<tr><td>57</td><td>abc...</td></tr>
<tr><td>58</td><td>aef...</td></tr>
<tr><td rowspan="2">Jogos</td><td rowspan="2">Jogo2</td><td rowspan="2">1001</td><td rowspan="2">34</td><td>ChaveB</td><td>ValorB</td></tr>
<tr><td>37</td><td>bbc...</td></tr>
</table>

Ilustração 20 - Exemplo de como se pode encontrar a Tabela de Mensagens Recebidas para Tratamento FIFO

A tabela dos contadores das mensagens FIFO contém um conjunto de contadores para marcar a sequência das mensagens enviadas que requerem tratamento FIFO, sendo a sua estrutura indicada na Ilustração 21.

Contadores para envio de mensagens com tratamento FIFO (_sendFIFOCounters)

Chave		Valor
Nome do Grupo (string)	Grupo de Ordenação (string)	Nº da última mensagem (long)

Ilustração 21 - Estrutura da Tabela de Contadores das Mensagens FIFO a enviar

3.6.2 Camada XTCPServer

A camada XTCPServer possui os módulos de software “Gestor de Servidor” e “Controlador TCP” (ver Ilustração 11). Esta camada é usada pelo servidor e tem como objectivo controlar o *socket* TCP para a comunicação e gerir o estado de servidor necessário à prestação de

serviços. Esta DLL possui um conteúdo que permite manipular o contexto do sistema de comunicações referente ao servidor assim como aceder aos seus serviços base. Esta classe tem várias estruturas de dados, que suportam os serviços que fornecem, das quais se destacam:

- A tabela com os grupos identificados pelo nome e com os clientes a eles associados;
- A tabela de clientes e com os grupos a que pertencem;
- A tabela dos pedidos que precisam de resposta.

Grupos (_groupsByName)

Chave	Valor
Nome do Grupo (string)	Lista de Clientes (List<long>)

Ilustração 22 - Tabela de Grupos

A tabela de grupos permite controlar os grupos que existem e saber qual a sua composição.
A estrutura da tabela de grupos tem a estrutura indicada na Ilustração 22.

Na chave desta tabela regista-se o nome do grupo. No valor correspondente está a respectiva lista de clientes associados ao grupo.

A tabela de clientes permite determinar em que grupos, um determinado cliente está registado. Esta informação consegue-se obter consultando a tabela de grupos através do varrimento do campo valor e consultando a sua chave (grupo). Para evitar a execução desta operação, optou-se por se criar uma tabela própria para fornecer, rapidamente, a informação sobre o cliente.
A tabela de clientes tem o formato indicado na Ilustração 23.

Esta tabela possui a lista de grupo associada a cada cliente. Tal como a tabela anterior, possui informação que é actualizada sempre que um cliente é adicionado ou é retirado de um grupo.

O servidor pede, com um período definido no ficheiro de configuração, que os clientes respondam se estão "vivos" (através do controlador "`Request_KeepAlive()`"). Se o cliente não responder a dois "KeepAlives" (configuração por omissão) consecutivos então é

eliminado de todos os grupos em que estiver activo, ou seja, as tabelas de grupos e de clientes são actualizadas.

Clientes (_groupsByClientId)

Chave	Valor
ID do Cliente (long)	Lista de Grupos (List<string>)

Ilustração 23 - Tabela de Clientes

Por fim, temos a tabela de pedidos com a estrutura indicada na Ilustração 24.

Pedidos (_requests)

Chave		Valor	
ID do Cliente emissor (long)	ID do Pedido do cliente emissor (long)	Mensagem (CMsg)	Controlador de Tempo (CRunTimeOut)

Ilustração 24 - Tabela de Pedidos do Servidor

Nesta estrutura o pedido é identificado pelo ID do cliente emissor e o ID do seu pedido. A mensagem e o controlador de tempo funcionam de forma idêntica à tabela de pedidos do emissor. A diferença fundamental é que, no caso da tabela de pedidos do cliente, se a mensagem for, por exemplo, destinada a outros três clientes, só uma mensagem é enviada ao servidor. O servidor por sua vez reenvia a mensagem aos três clientes destinatários e, quando receber a última resposta, devolve o resultado ao cliente eliminando a, respectiva, entrada da tabela de pedidos.

3.7 Servidor de Comunicações

O servidor de comunicações implementado (ver Ilustração 9), tem capacidade para responder simultaneamente a vários pedidos de ligação. Esta característica foi conseguida através da utilização de várias *threads* com suporte para comunicações assíncronas.

Este servidor tem um único *endpoint* TCP (IP e porto) que serve múltiplas ligações. Sempre que um novo cliente solicita uma ligação, o método TcpListener.BeginAcceptTcpClient do *namespace* System.Net.Sockets é executado dando início a uma operação assíncrona que usa uma *thread,* da *thread-pool* da plataforma .NET, que aguarda pela ligação do cliente. Assim, cada estabelecimento de ligação é tratado por uma *thread* (eventualmente diferente) *da thread-pool*. Quando o BeginAcceptTcpClient fica completo, chama o EndAcceptTcpClient para obter o novo objecto TcpClient.
Depois da ligação ser estabelecida, é criada uma nova *thread* para tratar do envio e recepção de dados da ligação em causa. Assim, não se consome *threads* da *thread-pool* por muito tempo e essa nova *thread* é terminada quando a ligação for terminada ou quando a *thread main* finalizar.

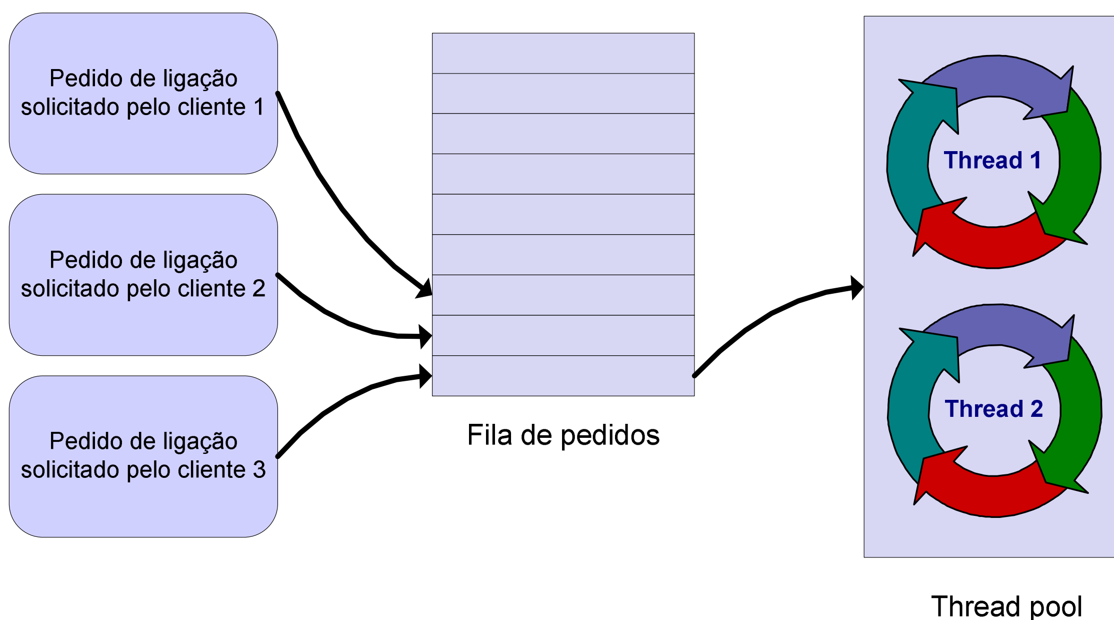


Ilustração 25 - Servidor de comunicações usando uma *Thread Pool*

Na Ilustração 25 visualizamos uma *thread pool* com duas *threads*. Quando chegam três pedidos eles são colocados numa fila a aguardar que sejam processados. Como as duas *threads* estão livres então os primeiros dois pedidos são processados. Assim que um desses pedidos seja satisfeito a respectiva *thread* fica livre e inicia o processamento do terceiro pedido. Neste cenário não houve necessidade de criar ou destruir *threads* da *pool* por razões de desempenho. Num cenário em que duas *threads* processam dois pedidos e o CPU não atinge 50% de utilização, estamos perante pedidos que aguardam por determinados eventos ou estão a executar operações I/O. Neste situação a *pool* pode detectar a disponibilidade do processador e incrementar o número de *threads* para que mais pedidos sejam satisfeitos ao mesmo tempo. Num cenário contrário em que o processador atinge os 100% de utilização, a *pool* diminui o número de *threads* para minimizar o tempo gasto em *context switches*.

4 Teste da Solução

Após o desenvolvimento do sistema de comunicações, houve necessidade de efectuar testes num cenário real. Pretendeu-se testar a comunicação entre máquinas pertencentes a redes diferentes com a presença de mecanismos NAT e onde os utilizadores estivessem distribuídos em grupos. As mensagens com e sem retorno de informação dos serviços de transporte de mensagens também foram verificadas.

Para concretizar o teste da solução projectou-se uma aplicação distribuída em que as diversas máquinas precisavam de trocar informações entre si. Esta aplicação consistiu num jogo didáctico, multiutilizador e distribuído, em que os participantes tinham de responder a questões sobre as matérias leccionadas no estabelecimento de ensino. As respostas certas são premiadas com uma pontuação e dessa forma distinguia-se a posição alcançada por cada jogador. Cada interveniente joga num PC e os seus adversários poderiam estar localizados na mesma rede ou em redes distintas. Esta aplicação persiste a informação numa base de dados e executa-se numa arquitectura segundo a Ilustração 26.

O servidor executa a Lógica de Negócio e a Lógica de Acesso a Dados enquanto o cliente corre a Lógica de Apresentação. A estas camadas de software foram adicionadas as camadas resultantes da solução do sistema de comunicações (ver ponto 3.6). A Ilustração 26 pretende mostrar a relação entre a arquitectura física e lógica da solução final (jogo mais o sistema de comunicações).

O ambiente de teste é composto por um servidor de dados, um servidor de comunicações (trata do *relaying* e gere a informação de grupos) e vários clientes. Entre cada cliente e o servidor de comunicações pode existir um NAT *Outbound* e uma *firewall*. As *firewalls* estão abertas de forma a permitir todo o tráfego da aplicação consumidora da plataforma de comunicações.

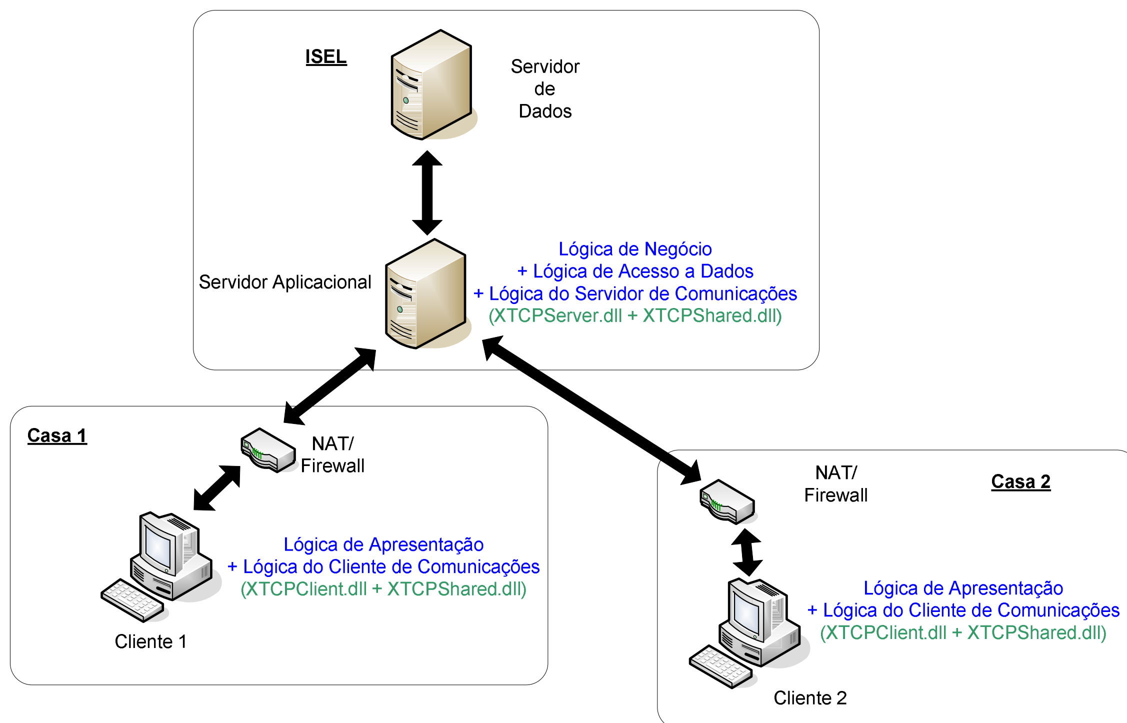


Ilustração 26 - Relação da Arquitectura Física e Lógica do Jogo

4.1 O Jogo Como Teste da Solução

O sistema de comunicações desenvolvido no trabalho foi, após ser concluído, testado em rede. Para que a utilização do sistema fosse parecida com um ambiente real, foi desenvolvida uma aplicação consumidora dos seus serviços. Essa aplicação consistiu num jogo didáctico cujos intervenientes poderiam estar localizados em diferentes redes privadas ou pública.

4.1.1 O Desenrolar de Um Jogo

O jogo desenvolvido, para testar a solução de comunicações, consiste em perguntar aos participantes questões previamente carregadas no sistema de informação que lhe estava associado. As questões são, através da aplicação cliente do jogo, previamente categorizadas e classificadas de escolha múltipla ou de desenvolvimento. Antes de um jogo começar, é necessário configurá-lo quanto ao número de perguntas a responder, às categorias a que devem pertencer as perguntas, quanto tempo se tem para responder, etc. O processo do jogador que configura o jogo é nomeado de coordenador desse jogo em particular. O coordenador tem a responsabilidade de gerir o jogo que iniciava, ao qual são adicionados os

elementos convidados. Se algum dos jogadores convidados desistir do jogo, este prossegue. Durante o decorrer do jogo, só há um jogador a responder num determinado instante. Os outros jogadores, vão visualizando a resposta, o tempo em falta e a posição do rato do jogador activo para terem uma ideia clara do que se passa na máquina de quem está a responder (perceber se há hesitação por parte do jogador ou se este se "inclina" para uma resposta aparentemente incorrecta).

Vejamos agora um exemplo de um jogo e a forma como a sua implementação tira partido dos serviços disponibilizados pela camada de comunicação e de como isso se traduz numa mais valia para o desenvolvimento de aplicações distribuídas.

A aplicação consumidora da infra-estrutura de comunicações, neste caso é o jogo, não precisa de implementar novos controladores de mensagens a não ser que pretenda estender os seus serviços. Um exemplo de um serviço novo que pudesse fazer sentido, seria a criação de vários grupos de utilizadores a uma determinada hora com um único pedido. De acordo com as funcionalidades base do sistema de comunicações, a criação de cada grupo é feito através de um serviço cujo método *entry point* designa-se por «`Request_CreateGroup`». A sua sintaxe é
«`bool? Request_CreateGroup(string groupName)`».
No entanto este novo serviço sugere uma sintaxe do tipo
«`void Request_CreateGroup(string[] groupNames, DateTime dt)`».

Contudo, quase todos os tipos de mensagens necessários à lógica do jogo foram implementados através dos serviços "`Request_SendData`" e "`Request_SendDataAndReceiveAnswer`". Estes serviços são os responsáveis por fazer chegar os dados de um cliente emissor a um conjunto de clientes destinatários, com a particularidade que o primeiro serviço «transporta» avisos enquanto o segundo controlador «conduz» pedidos (existe mensagem de retorno).

Vejamos agora um exemplo de como se poderia desenrolar um jogo. Vamos supor que o jogador Jorge Santos está registado no sistema e que iria iniciar um jogo. O registo no sistema é feito com o estabelecimento da ligação TCP entre o cliente e o servidor. A acção de registo é um dos serviços de iniciação disponibilizados pela plataforma de comunicações. Após o registo, o cliente executa o serviço "`Request_WhoAmI`" para obter o seu ID único perante o sistema de comunicações.

De seguida o jogador pretende dar início a um jogo colectivo e, para isso, selecciona a opção adequada. Consequentemente, aparece um formulário onde se configura o jogo que pretende iniciar conforme Ilustração 27.

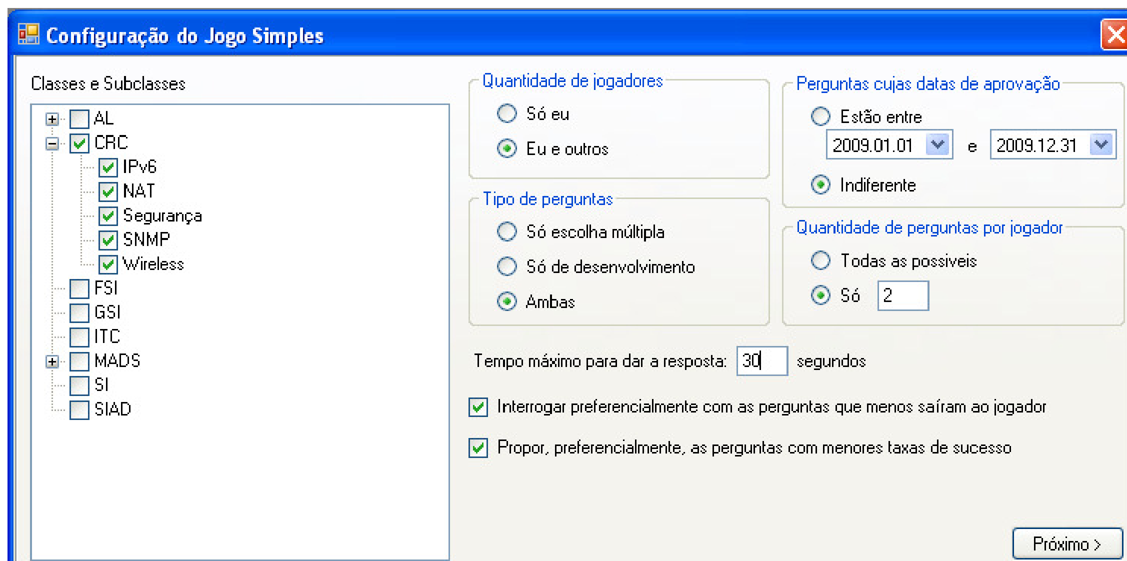


Ilustração 27 - Ecrã de Configuração do Jogo Simples

Depois de pressionar o botão "Próximo", e tendo em consideração que é um jogo colectivo, o processo coordenador do grupo (quem inicia e configura o jogo) faz um *broadcast* a perguntar quem é que está disponível para jogar e pergunta também informação sobre o jogador (como, por exemplo, o nome e o ID). De recordar que qualquer mensagem é feita via servidor. Portanto, no caso desta mensagem, ela foi transferida para o servidor que a reenviou a todos os elementos do grupo "Main" (grupo com todos os elementos activos), aguardou resposta de todos os clientes e enviou o resultado para o coordenador (cliente emissor).

O serviço usado para a mensagem em causa é o "`Request_SendDataAndReceiveAnswer`". Durante este procedimento, o ecrã do coordenador fica com a imagem da Ilustração 28. Como resultado obtém-se um formulário com os jogadores disponíveis (jogadores activos e que não estão a jogar), conforme a Ilustração 29.

De seguida o jogador Jorge Santos escolhe dos jogadores que estão disponíveis aqueles que pretende convidar para o jogo. Para isso marca os jogadores em causa e pressiona o botão "Próximo" conforme Ilustração 29. Nesta fase é criado um grupo no qual vão ser adicionados todos os elementos convidados e que queiram participar no jogo. Para criar o grupo usa-se o serviço "`Request_CreateGroup`".

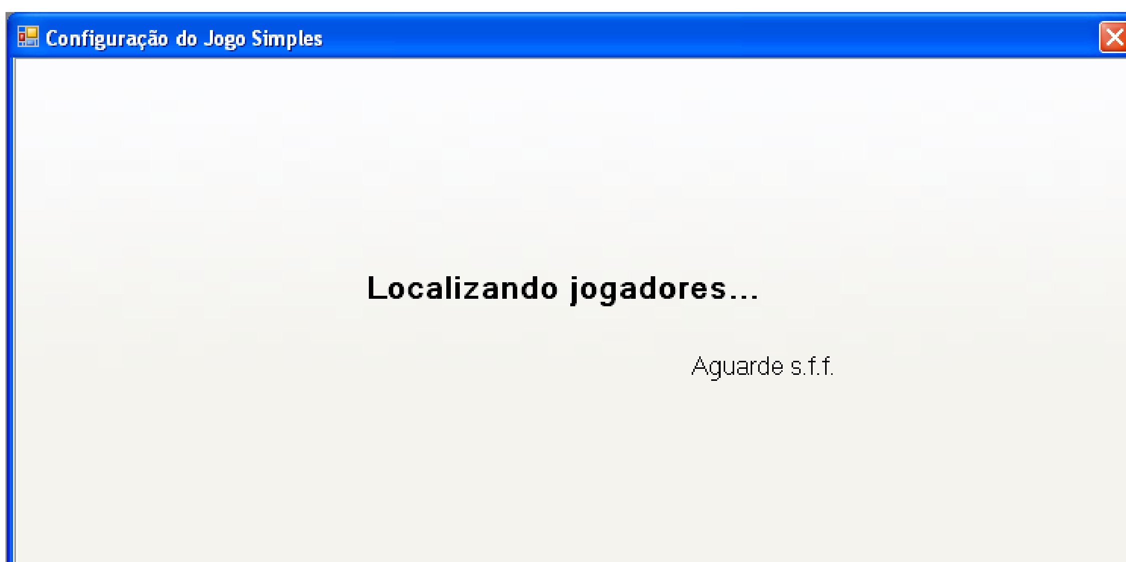


Ilustração 28 - Ecrã de indicação de que se está a localizar jogadores disponíveis.

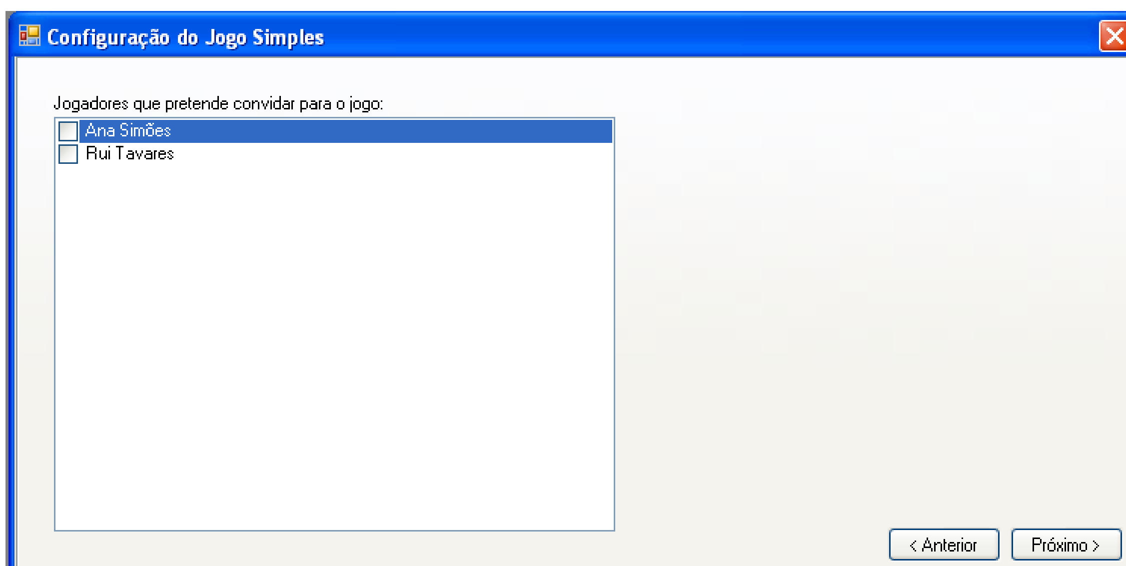


Ilustração 29 - Ecrã com indicação dos jogadores disponíveis.

Vamos supor que o jogador convidado foi o Rui Tavares.

Assim, no ecrã do Jorge Santos aparece indicação de que está a fazer um convite (Ilustração 30), enquanto no ecrã do Rui Tavares aparece o convite (Ilustração 31). A acção de convidar que parte do jogador Jorge Santos é feita de forma assíncrona uma vez que a resposta pode demorar vários segundos e é necessário conseguir refrescar o ecrã ou mesmo desistir do convite (e daí a *thread* não poder ficar bloqueada). O serviço usado para o convite foi o "`Request_SendDataAndReceiveAnswer`" mas com um ajuste na sua propriedade de *timeout* no sentido de aceitar uma resposta que pode demorar. Ao chamar este serviço

indica-se uma lista de destinatários para os convites, o nome do grupo a que deverão ser adicionados se aceitarem o convite, quanto tempo têm para responder ao convite e a mensagem de convite;

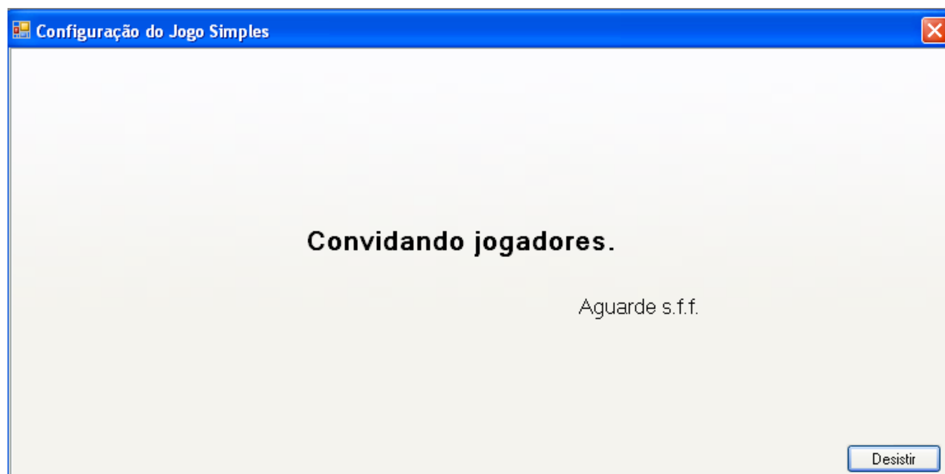


Ilustração 30 - Ecrã com indicação de que o convite está a ser feito (computador de Jorge Santos)

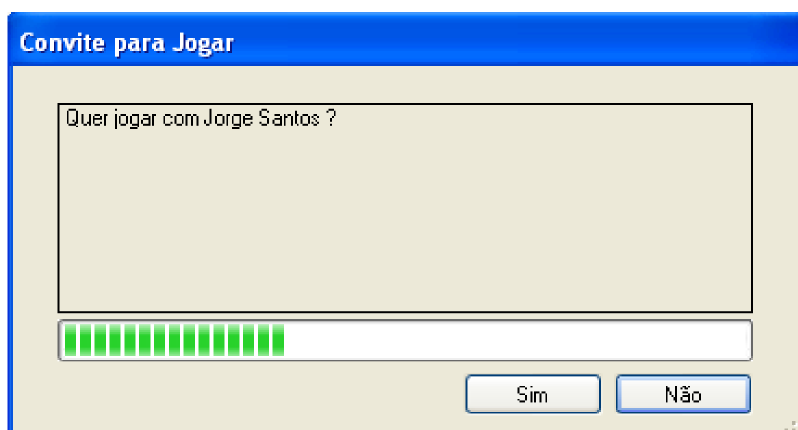


Ilustração 31 - Ecrã com indicação do convite (computador de Rui Tavares)

O jogador Rui Tavares ao receber o convite - Ilustração 31 tem um determinado tempo para responder. Se não responder dentro desse período de tempo (tempo configurável e que por omissão é de 10 segundos) a aplicação irá assumir que não aceitou o convite.

Vamos supor que o Rui Tavares aceitou o desafio.

No ecrã do coordenador irá aparecer a lista de jogadores - Ilustração 32 que irão participar no jogo, enquanto o Rui Tavares fica com indicação de que o jogo está a ser iniciado - Ilustração 33.

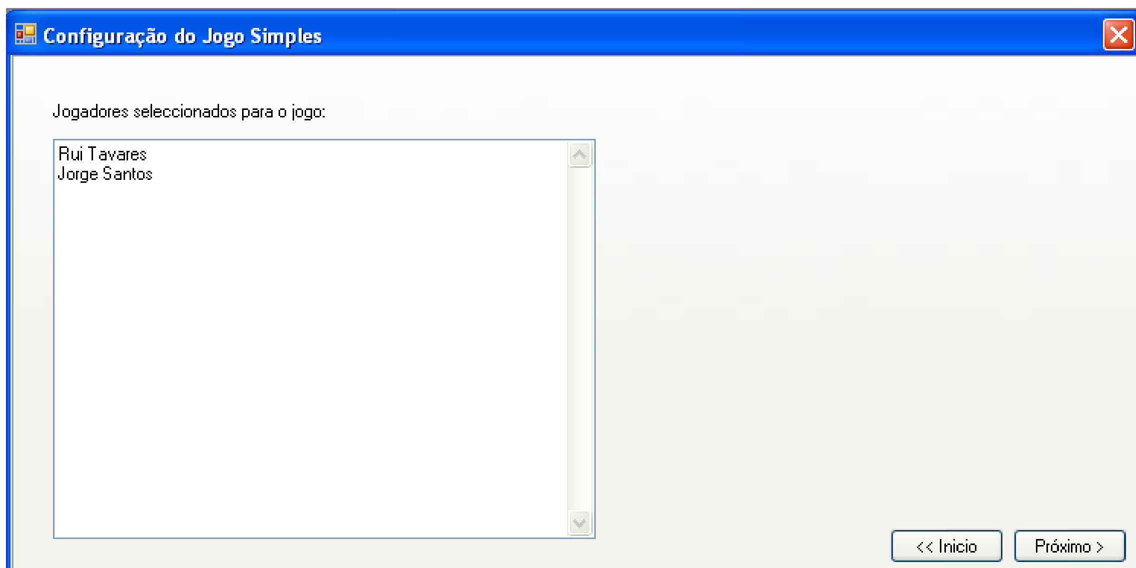


Ilustração 32 - Lista de jogadores seleccionados para o jogo (computador do Jorge Santos)

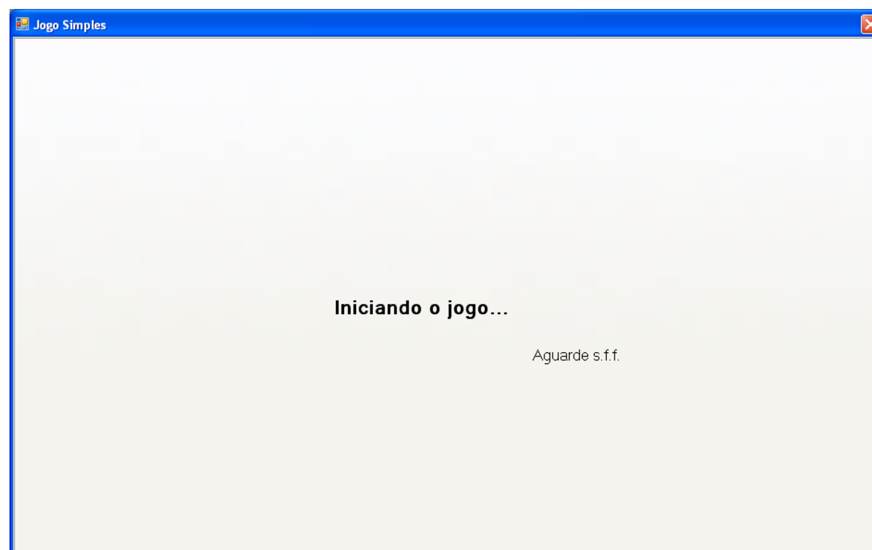


Ilustração 33 - Indicação de que o jogo vai começar (computador do Rui Tavares)

Assim que o jogador Jorge premir o botão “Próximo”, o coordenador do grupo vai pedir perguntas à base de dados de acordo com a configuração indicada inicialmente (como por exemplo, interrogar preferencialmente com as perguntas que menos saíram ao jogador). As perguntas seleccionadas serão dispostas aleatoriamente assim como a ordem pela qual os jogadores serão interrogados. Os jogadores serão interrogados sequencialmente e quando um jogador estiver a ser interrogado os outros jogadores visualizam o que o jogador activo estiver a fazer (vêem a pergunta que foi feita, a resposta que está a ser dada no caso de uma questão de desenvolvimento, etc.). O coordenador irá executar mais um serviço de “`Request_SendDataAndReceiveAnswer`” que irá iniciar o estado em todos os jogadores seleccionados (o estado comporta entre outras coisas os nomes dos jogadores e os

seus identificadores). Depois será solicitado outro serviço de "Request_SendDataAndReceiveAnswer", com o objectivo de transferir para os participantes a próxima pergunta a fazer, as respectivas respostas, o número da pergunta e a pontuação actual dos jogadores. Por fim é executado um serviço de "Request_SendData", em que o coordenador informa os intervenientes sobre o nome do jogador activo e a sua identificação (entenda-se por jogador activo aquele que irá responder).

Assim que estiver tudo preparado aparece o ecrã da Ilustração 34 em todos os jogadores, a indicar quem irá responder à questão corrente. Só no formulário do jogador activo é que irá aparecer o botão "Avançar".

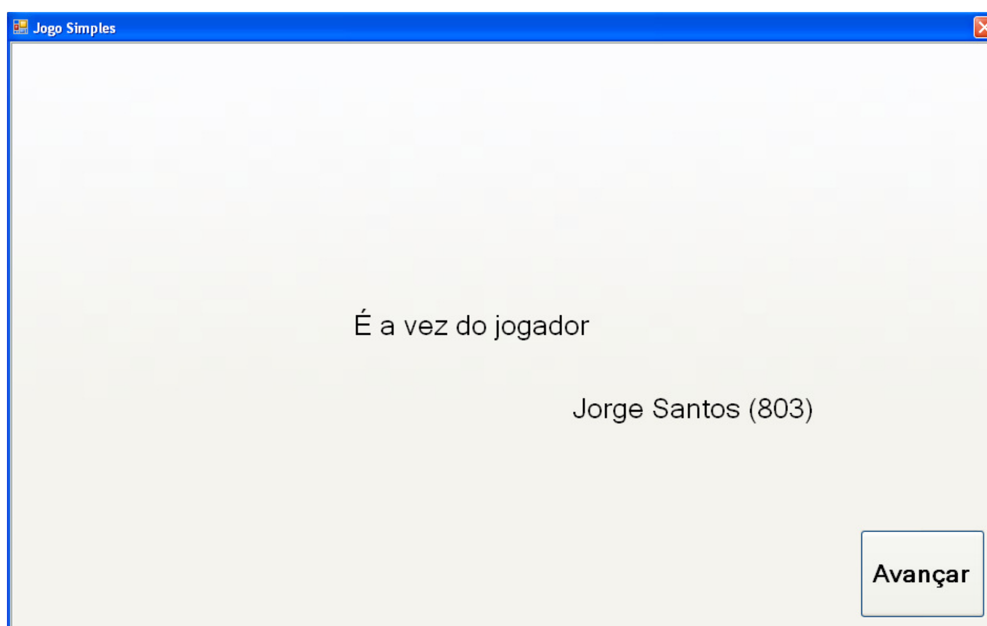


Ilustração 34 - Ecrã a dar indicação de quem irá jogar de seguida. Nota o botão "Avançar" só aparece no jogador activo.

Quando o Jorge premir o botão "Avançar" irá aparecer um ecrã com a pergunta em todos os jogadores - Ilustração 35. No entanto só o Jorge terá os botões de selecção de resposta activos. O serviço com o controlador da mensagem "Request_SendDataAndReceiveAnswer" é executado para enviar a mensagem aos outros jogadores de que é preciso mudar de ecrã e ajustá-lo de acordo com a pergunta, respostas e tipo de pergunta (escolha múltipla ou de desenvolvimento). Nesta fase os jogadores passivos mudam para o ecrã da Ilustração 36.

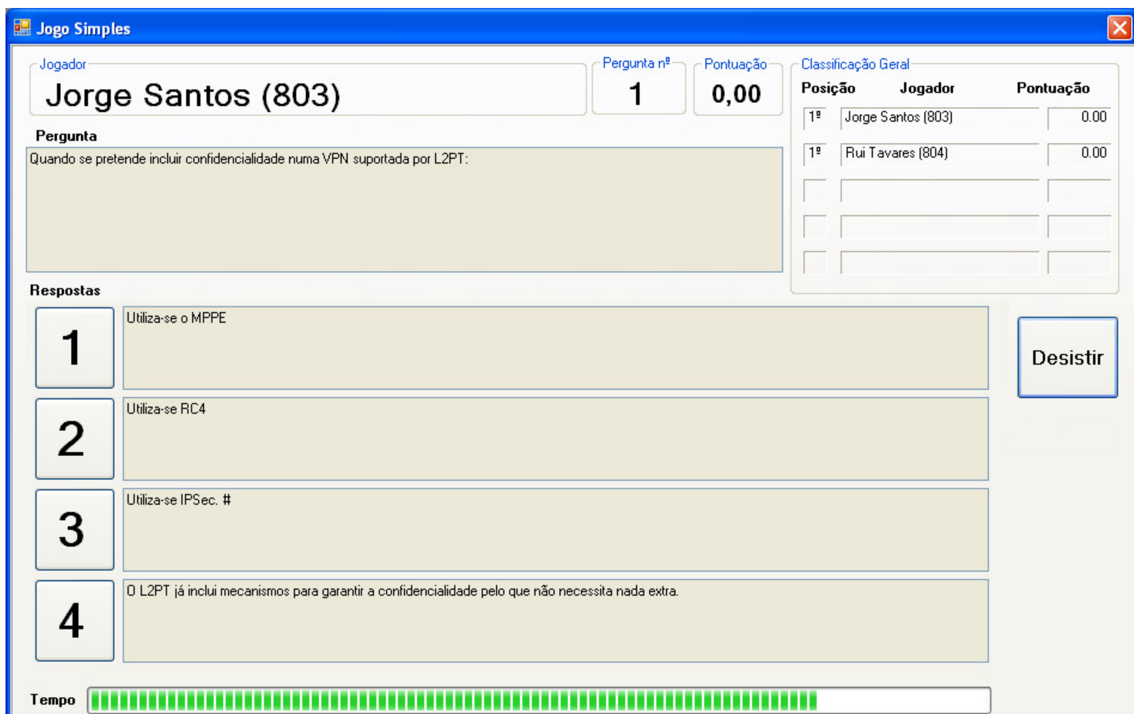


Ilustração 35 - Ecrã em que o jogador activo está a responder a uma questão de escolha múltipla (computador do Jorge)

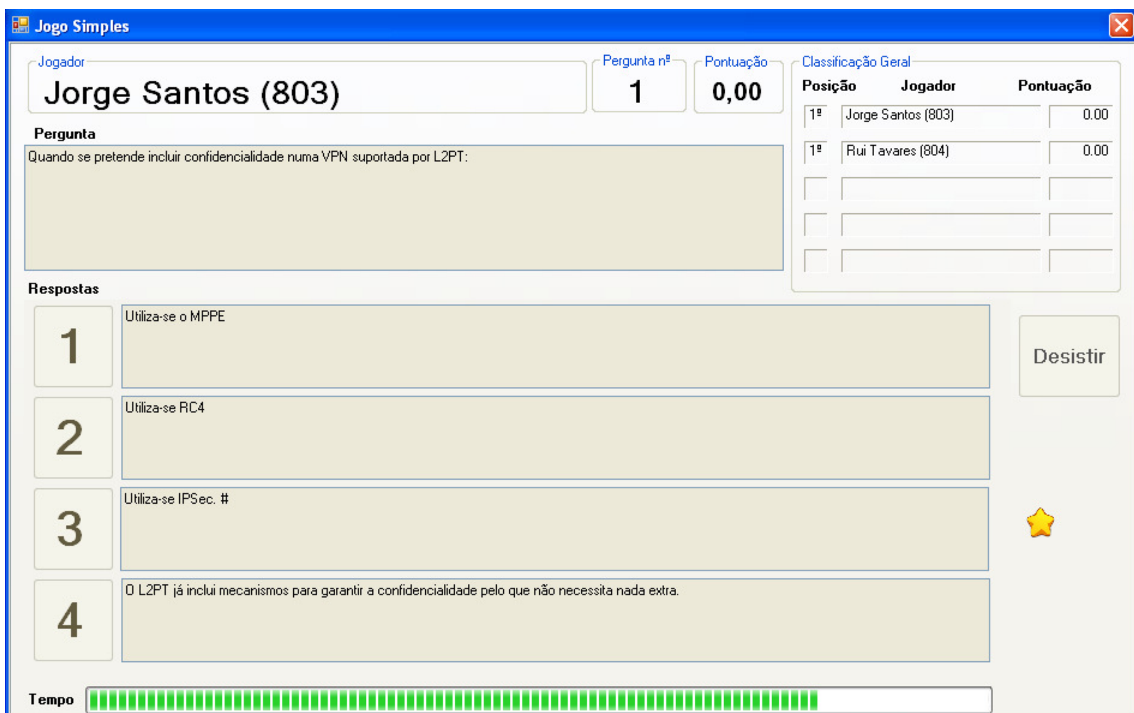


Ilustração 36 - Ecrã em que o jogador passivo assiste a uma resposta de escolha múltipla (computador do Rui). A estrela indica a posição do rato no ecrã do jogador activo.

Depois de aparecer o ecrã para dar a resposta, o jogador activo envia periodicamente informação aos outros jogadores. Essa informação é enviada através de mensagens do serviço "`Request_SendDataAndReceiveAnswer`" e consiste na indicação da posição do rato,

o tempo gasto e a resposta dada até ao momento (útil no caso de pergunta de desenvolvimento). Desta forma os jogadores passivos sabem tudo o que se está a passar com o jogador activo, mesmo a sua posição do rato que é representado com uma estrela amarela ou por um *smile*. A ideia é que todos os jogadores possam exercitar os seus conhecimentos mesmo quando não são o jogador corrente. O facto de se saber qual a posição do rato do jogador activo permite prever se a resposta que ele vai dar corresponde à expectativa do jogador passivo.

Quando o jogador activo selecciona uma resposta, prime "Desistir" ou o tempo para responder acaba, os outros jogadores são informados através de mensagens do método "`Request_SendDataAndReceiveAnswer`", do número da resposta dada ou o texto da resposta (caso seja uma pergunta de desenvolvimento), o tempo gasto na resposta, os identificadores da pergunta e resposta. Desta forma, os controlos visuais são ajustados assim como os quadros de classificação. O jogador Jorge ao seleccionar a resposta número três obtinha o ecrã da Ilustração 37.

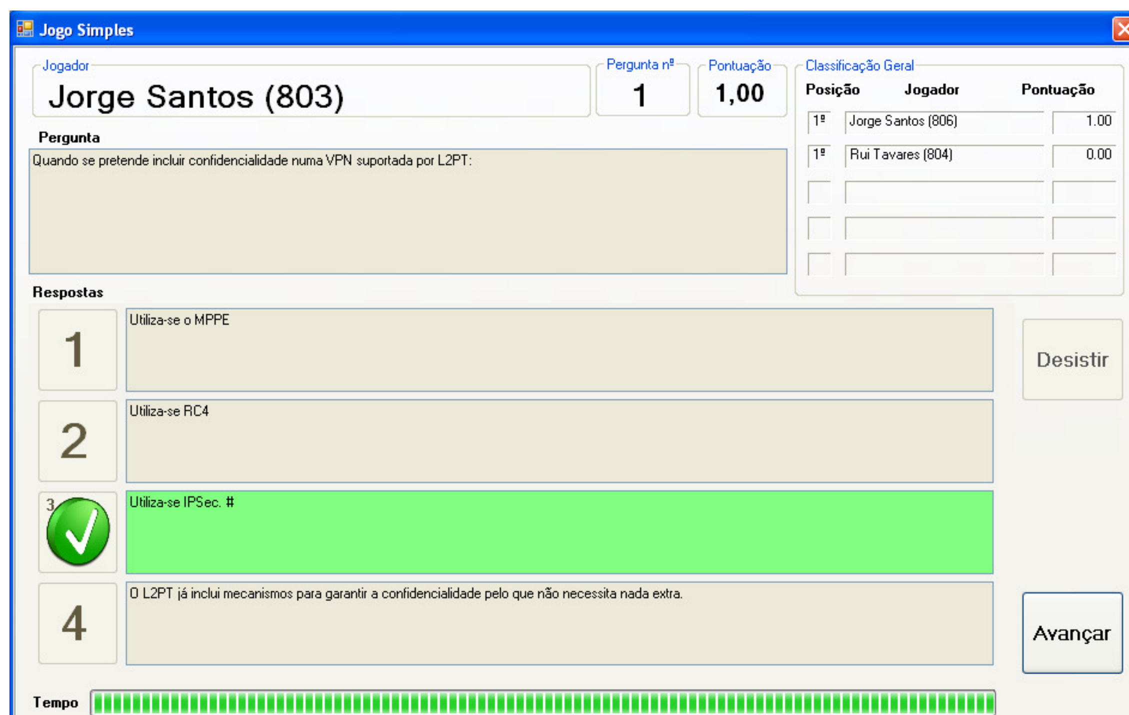


Ilustração 37 - Ecrã obtido após selecção da resposta correcta

A resposta correcta surge com o fundo a verde e o botão seleccionado aparece com um visto porque acertou na solução.

Ao premir "Avançar" (este botão só aparece no ecrã do jogador activo) o coordenador do jogo é avisado com uma mensagem do serviço "`Request_SendData`", que lhe solicita a

identificação do próximo jogador. A partir deste momento a solução repete os passos ocorridos desde a execução do serviço que transferiu a próxima pergunta para os jogadores. Ou seja, é transferido para os participantes a próxima pergunta a fazer, as respectivas respostas, o quadro da pontuação actual dos jogadores e o número da pergunta corrente. Depois é executado mais um serviço "`Request_SendData`", em que o coordenador informa os intervenientes sobre o nome do jogador activo e a sua identificação. Volta-se, assim, ao ecrã da Ilustração 34 mas para outro jogador activo.

Supondo que a próxima pergunta era de desenvolvimento iria aparecer um ecrã com algumas alterações.

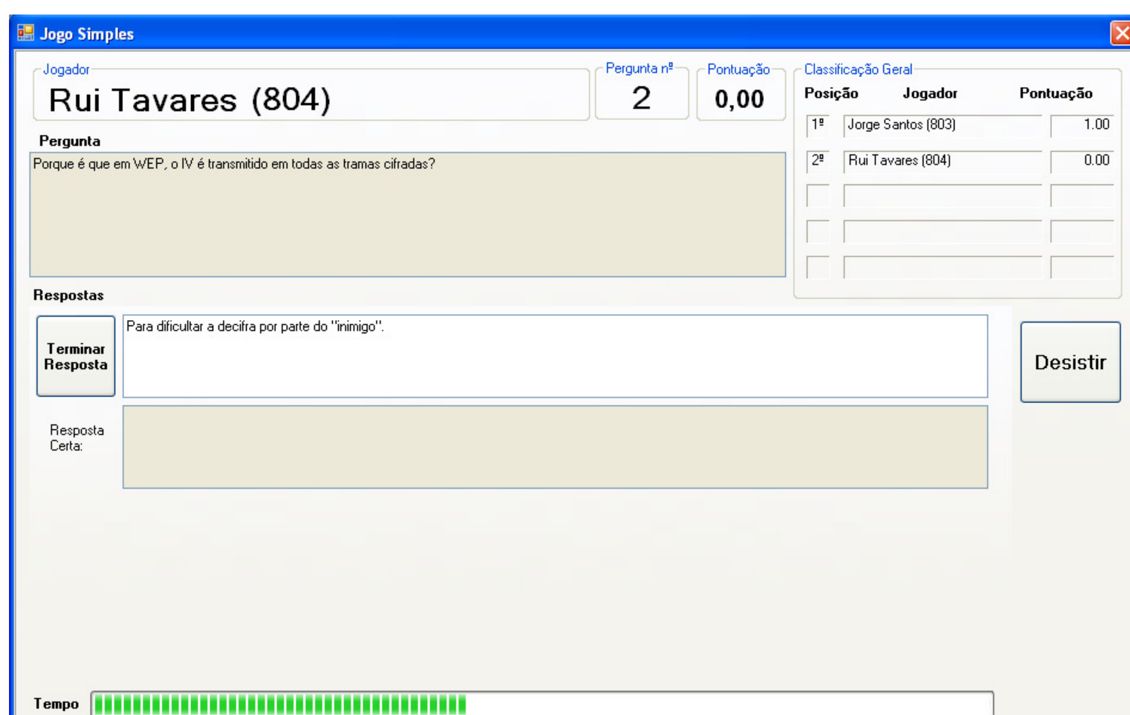


Ilustração 38 - Exemplo de um ecrã em que a resposta não é de escolha múltipla.

Quando o Rui premir o botão "Terminar Resposta" é enviada uma mensagem através do servido "`Request_SendDataAndReceiveAnswer`" que dá indicações aos outros jogadores de que a resposta foi dada. Como é uma resposta de desenvolvimento então todos os jogadores passivos vão dar uma classificação à resposta - Ilustração 39. Para facilitar a atribuição de cotação é mostrada uma resposta correcta. A pontuação final atribuída à resposta será igual à média ponderada das classificações. O jogador activo visualiza no ecrã as classificações que são dadas pelos outros jogadores - Ilustração 40.

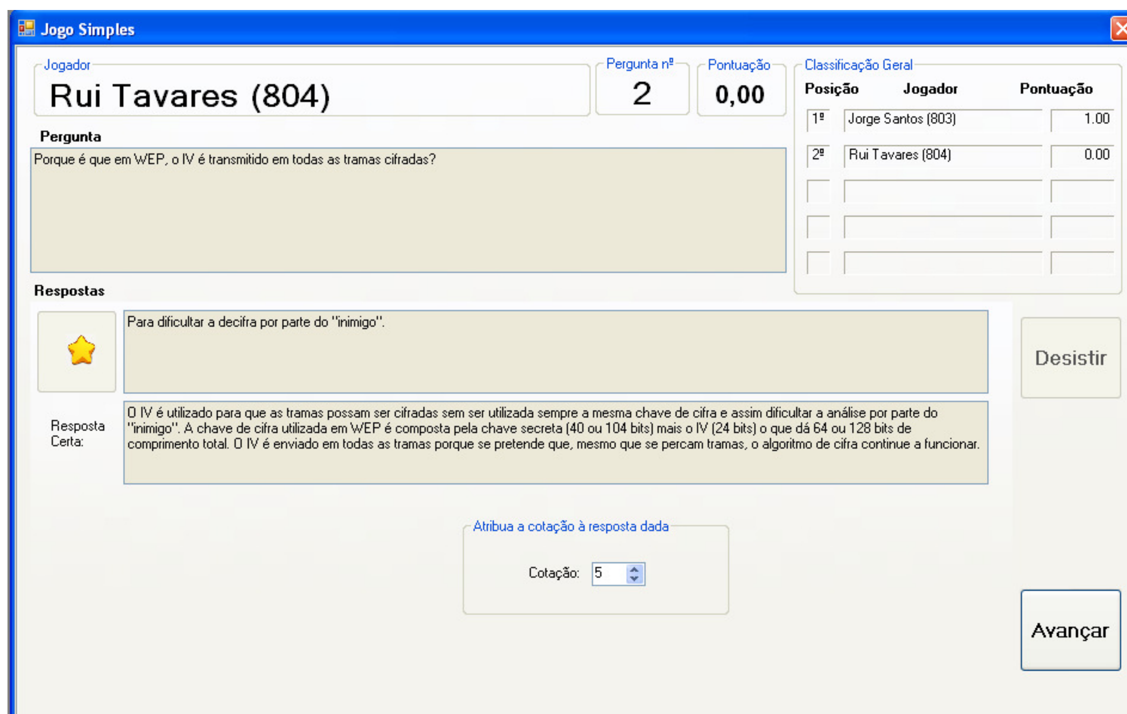


Ilustração 39 - Ecrã que o jogador passivo recebe para classificar a resposta do jogador activo

Ilustração 40 - Ecrã que o jogador activo visualiza após responder a uma pergunta de desenvolvimento

Quando o jogador passivo atribuir a classificação e pressionar "Avançar", será enviada uma mensagem para todos os jogadores, através do serviço "`Request_SendData`", a informar sobre a pontuação atribuída.

Assim, que o jogador activo receber todas as pontuações dos outros jogadores fica com o botão "Avançar" visível e o jogo pode prosseguir - Ilustração 41.

Ilustração 41 - Ecrã com as classificações atribuídas à resposta de desenvolvimento.

Ao premir o botão "Avançar" o coordenador do jogo é informado por uma mensagem do serviço "`Request_SendData`" que solicita que se passe ao jogador seguinte. Desta forma o jogo vai-se desenrolando até que se esgote as perguntas.

De referir que sempre que se muda de pergunta, o coordenador do jogo, solicita de forma assíncrona, que o resultado da resposta seja persistido. Desta forma consegue-se seleccionar questões com determinados critérios indicados na configuração do jogo como é o caso de "Propor, preferencialmente, as perguntas com menores taxas de sucesso" e "Interrogar preferencialmente com as perguntas que menos saíram ao jogador".

Quando o jogo termina, uma mensagem do serviço "`Request_SendData`" é enviada para todos os jogadores. Nessa mensagem consta a informação necessária à elaboração do quadro geral da pontuação final - Ilustração 42.

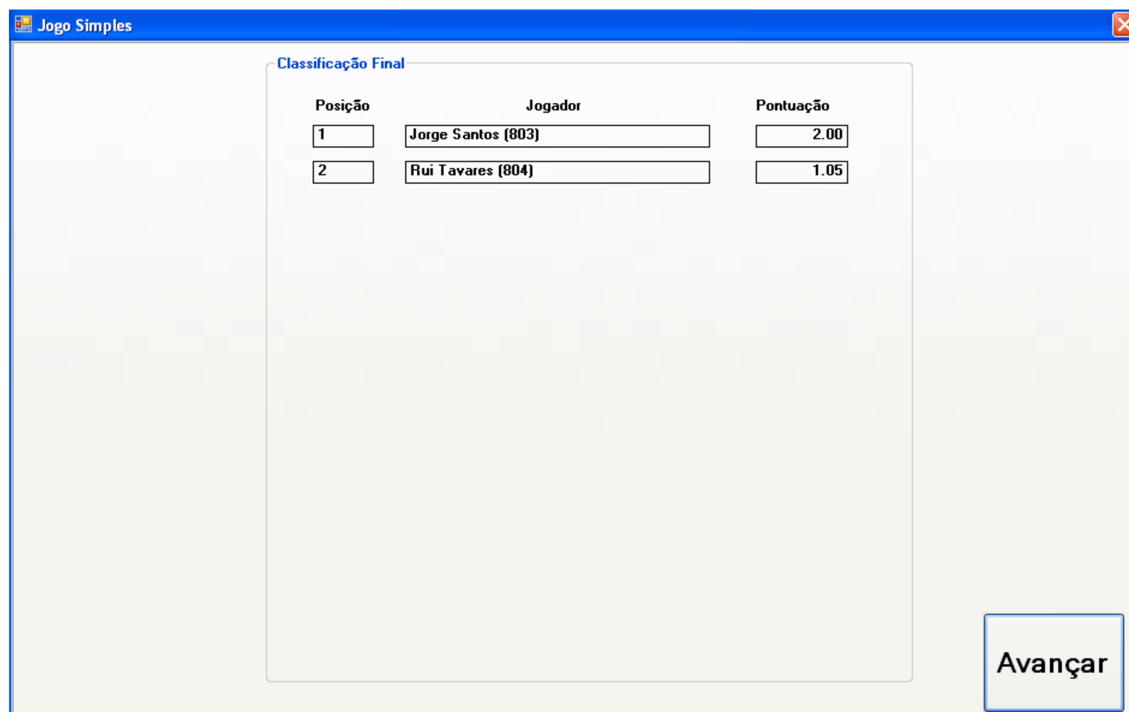


Ilustração 42 - Ecrã com a classificação final

A utilização do sistema de comunicações levou a que a implementação do jogo ficasse bastante simplificada, pois, todas as comunicações são geridas pela plataforma. As acções de trocas de mensagens solicitadas pelo jogo não dependem da localização dos PCs intervenientes, não têm de gerir as *streams* dos *sockets* TCP nem têm de se preocupar com a difusão das mensagens *broadcast* ou *multicast*. Todos estes serviços são fornecidos pela plataforma de comunicações o que facilita bastante o desenvolvimento de aplicações distribuídas. O requisito base para que o sistema funcione é os clientes conseguirem chegar ao servidor de comunicações e este estar operacional.

5 Conclusões

5.1 Plataforma de Comunicações Desenvolvida

Realizou-se um sistema de comunicações de apoio a aplicações distribuídas, capaz de ultrapassar as dificuldades inseridas por *firewalls* e NATs, utilizando-se a Framework .NET. Para testar o sistema desenvolvido elaborou-se uma aplicação distribuída baseada em jogos didácticos, cujos utilizadores poderiam estar localizados em redes distintas privadas ou pública.

A plataforma de comunicações, que se desenvolveu, fornece um conjunto de serviços base para controlo de grupos. Entre esses serviços encontram-se a definição da composição dos conjuntos e avisos aos elementos de um grupo sempre que ele sofra alterações. A plataforma pode ser estendida de forma a incorporar novos serviços específicos, e, consequentemente, ser utilizada noutro tipo de aplicações.

A forma de usar a plataforma de comunicações consiste em distribuir duas DLLs com a aplicação cliente e duas DLLs com a componente servidora da aplicação (senda uma das DLLs igual no cliente e servidor). Após a inclusão destes *assemblies*, deve-se subscrever os eventos necessários à concretização de alguns serviços automáticos (serviços de notificação) e inserir as chamadas de iniciação e finalização do servidor e cliente. Estes *assemblies* permitem que o programador se abstraia do tratamento das comunicações (não tem que gerir *streams* de *sockets*), mesmo em máquinas localizadas em redes diferentes, e não têm que se preocupar com a difusão de mensagens *broadcast* ou *multicast*, o que simplifica muito o desenvolvimento da solução distribuída pretendida.

5.2 Trabalho Futuro

Um dos pontos que pode ser melhorado é a escalabilidade do servidor de comunicações. Este servidor, com funções de *relay* é também um repositório central da informação sobre a composição dos participantes nas comunicações. Por esta máquina passam todas as comunicações. Uma abordagem possível para tornar a carga das comunicações escalável e/ou

ter redundância era ter uma estratégia semelhante à usada no Messenger. Aplicando essa abordagem teríamos um servidor com um IP público fixo ou com o nome constante (sendo o endereço resolvido por DNS) ao qual todos os utilizadores iniciavam uma ligação. Desta ligação haveria uma troca de mensagens para determinar qual o servidor de comunicações, que pertenceria a uma *pool*, se encontrava mais disponível. Depois do utilizador saber qual o IP do servidor de comunicações que estava mais livre, iniciava uma ligação para esse servidor. Assim, haveria balanceamento de carga entre os servidores de comunicação dessa *pool*. Quando o utilizador *A* quisesse contactar o utilizador *B* fazia-o através do seu servidor de comunicações que podia ser igual ou diferente do servidor de *B*. Os servidores de comunicações estabeleciam entre si as respectivas "pontes" para permitir as trocas de informações entre os utilizadores.

Outra abordagem possível era tentar efectuar uma ligação TCP entre os participantes, aplicando uma arquitectura *peer-to-peer*, e, só quando não fosse possível, usava uma *pool* de servidores para *relaying* com uma topologia cliente-servidor. Para a tentativa do estabelecimento de uma ligação "directa" entre os participantes poderia utilizar-se uma técnica de *Hole Punching*. A arquitectura *peer-to-peer* teria de ser híbrida, pois precisávamos de guardar algures a informação para controlo dos grupos.

Quanto à escalabilidade do estado, se ele estiver concentrado numa máquina, pode-se utilizar técnicas de fragmentação e/ou replicação dos dados. Em caso de fragmentação horizontal de dados, poder-se-ia agrupar a informação dos conjuntos, por exemplo, por ranges de IPs clientes. Aplicando, adicionalmente, uma técnica de transparência de fragmentação o utilizador não necessita de saber como é que os dados estão fragmentados. A utilização de replicação dos dados permite a existência de várias bases de dados cujos conteúdos se vão sincronizando regularmente proporcionando escalabilidade. Utilizando, adicionalmente, uma técnica de transparência de localização leva a que os participantes não precisem de conhecer a localização física dos dados. Esta técnica pode ser implementada com uma *pool* se servidores *stateless* de primeira linha que reencaminham os dados e direccionam as consultas para os servidores finais de dados (estando estes com dados eventualmente fragmentados e ou replicados).

6 Referências Bibliográficas

[1] "P2P," http://pt.wikipedia.org/wiki/P2P, [2009.06.20, 2009].

[2] G. Venkatachalam, "Developing P2P protocols across NAT," *Linux Journal* vol. August 2006, no. 148, 2006.

[3] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-Peer Communication Across Network Address Translators."

[4] J. Kuthan, "Internet Telephony Traversal across Decomposed Firewalls and NATs," *Proceedings of the 2nd IP Telephony Workshop*, April 2001, 2001.

[5] W. R. Stevens, *TCP/IP Ilustrated, Volume 1*: Addison Wesley, 2004.

[6] B. Traversat, M. Abdelaziz, M. Duigou *et al.*, "Project JXTA Virtual Network," *Sun Microsystems, Inc.*, 2002.

[7] "JXTA(TM) Community Project," 2008.

[8] "JXTA Java™ Standard Edition v2.5: Programmers Guide," I. Sun Microsystems, ed., 2007.

[9] "Skype," http://www.skype.com.

[10] S. A. Baset, and H. Schulzrinne, "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol," *IEEE INFOCOM*, September 15, 2004, 2004.

[11] S. Guha, N. Daswani, and R. Jain, "An Experimental Study of the Skype Peer-to-Peer VoIP System," in The 5th International Workshop on Peer-to-Peer Systems, 2006.

[12] J. Schmidt. "The hole trick," http://www.h-online.com/security/features/How-Skype-Co-get-round-firewalls-747197.html.

[13] M. Mintz, and A. Sayers. "MSN Messenger Protocol," http://www.hypothetic.org/docs/msn/index.php.

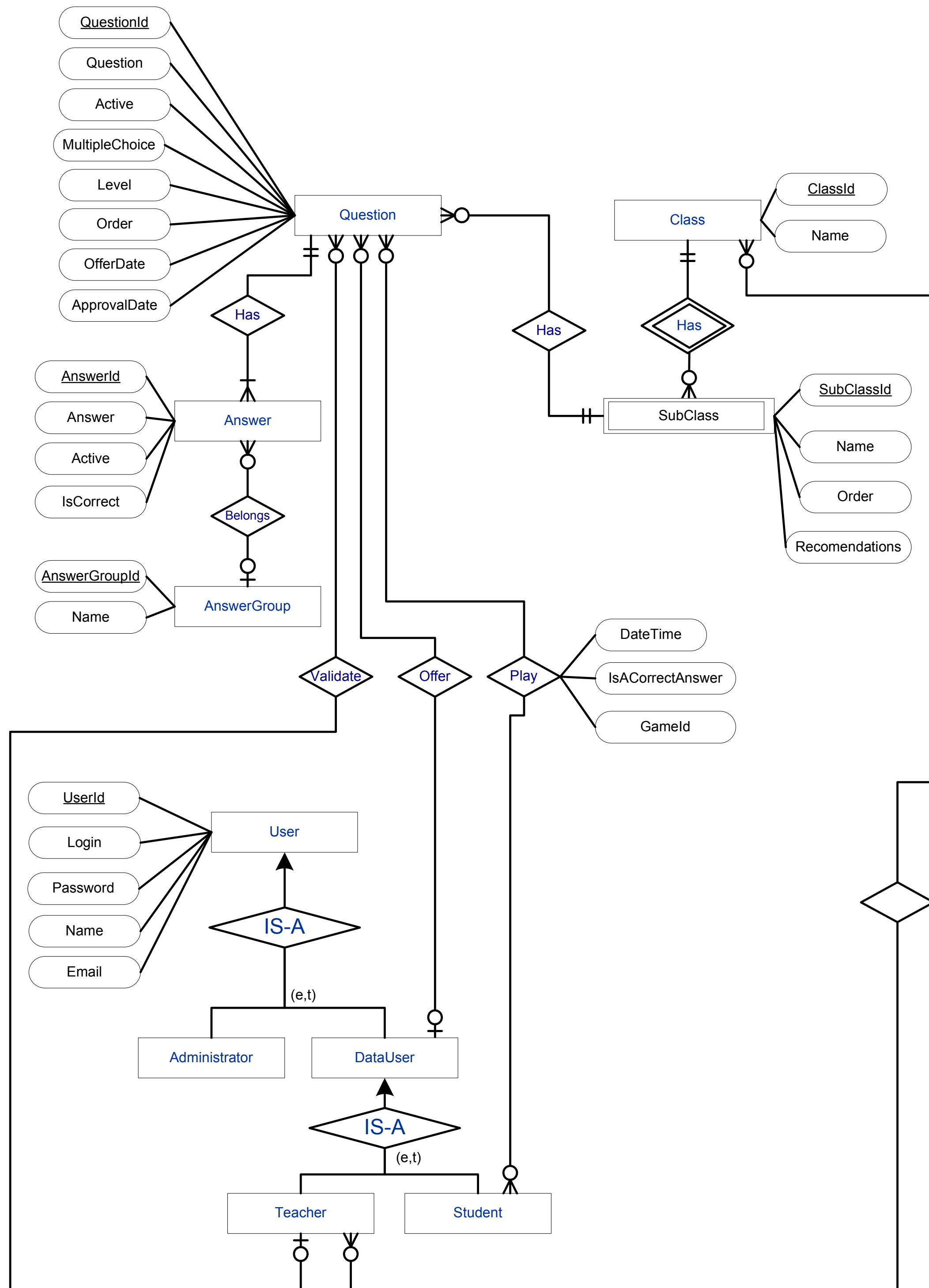
[14] V. Roca, and A. El-Sayed, *A Host-Based Multicast (HBM) Solution for Group Communications*, p.^pp. 610-619: Springer Berlin / Heidelberg, 2001.

[15] L. A. B. Estefanel, "Estudo sobre Comunicação de Grupos para Tolerância a Falhas," *Proceedings of the IV Simpósio Nacional de Informática*, 1999.

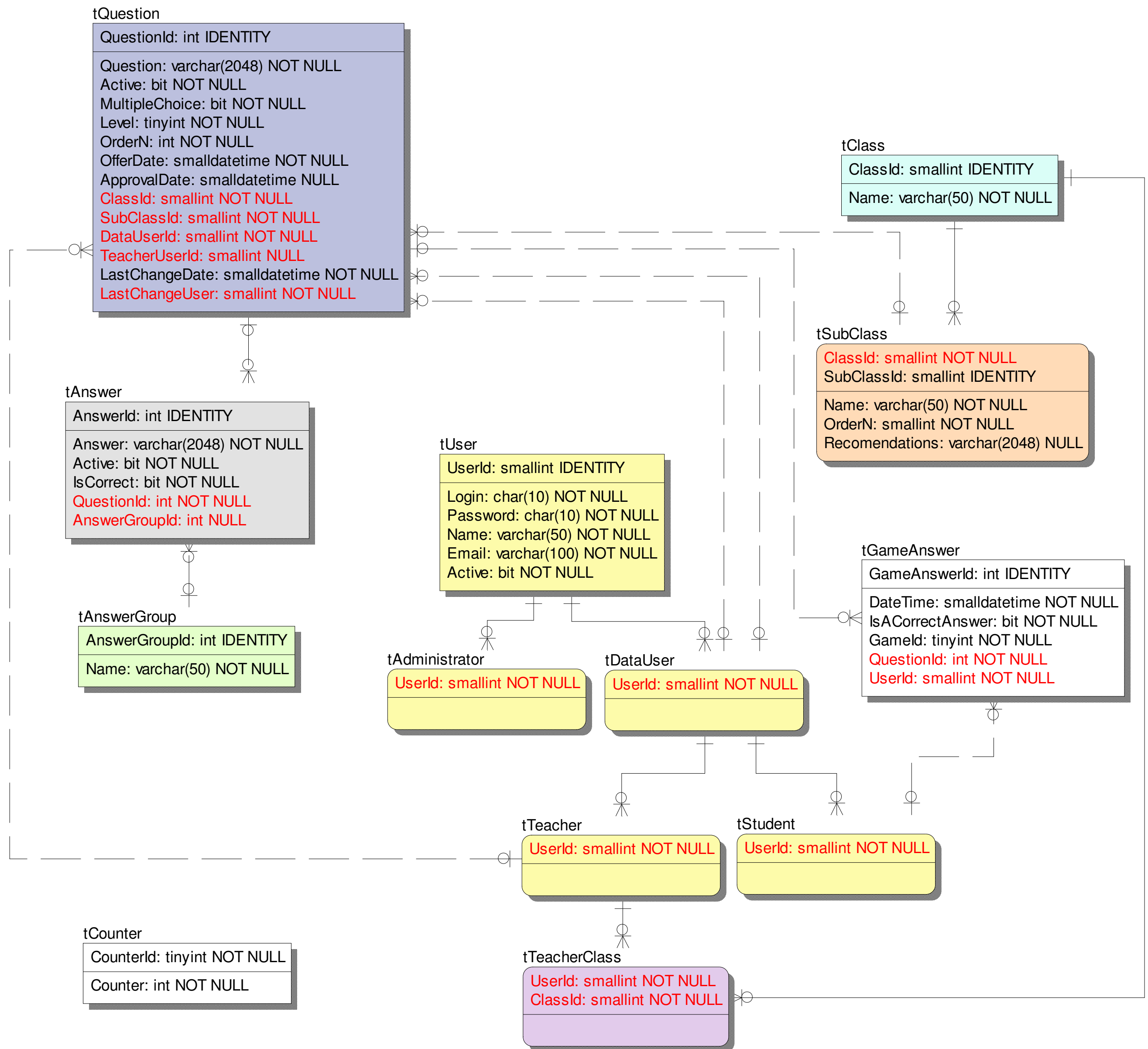
[16] A. El-Sayed, and V. Roca, "A Survey of Proposals for an Alternative Group Communication Service," *IEEE Network,* vol. 17, no. 1, pp. 46-51, 2003.

Anexo A – Modelo de Dados do Jogo

A.1 Modelo Entidade-Associação



A.2 Modelo Relacional



Anexo B - Interacção do jogo com o seu sistema de informação

A execução de um jogo usa um conjunto de dados que estão armazenados num sistema de informação. Antes da execução dos primeiros jogos é necessário carregar a base de dados com algumas perguntas e respostas, segundo determinados critérios. Para isso é necessário criar utilizadores do sistema, cujo perfil determina as respectivas permissões.

Quando se entra no sistema com um login de administrador pode-se criar outros utilizadores e listar os utilizadores que estão registados no sistema. Qualquer utilizador pode alterar a sua senha ou fazer *logoff*.

Vejamos agora um exemplo de um formulário para criar um utilizador já preenchido com os dados para um novo professor que lecciona ITC (Introdução à Teoria de Circuitos).

Ilustração 43 - Formulário para inserir um novo utilizador

Depois de premir “Ok” a aplicação cliente envia os dados para a aplicação servidora com a lógica aplicacional. Esta por sua vez executa as suas validações de negócio, como por exemplo, se o login já existe no sistema. Se estiver tudo correcto então os dados são enviados para a camada de acesso a dados que irá tratar da sua persistência. Este é o fluxo típico para os dados a inserir no sistema de informação.

No caso do utilizador autenticado ser um professor, pode criar classes e subclasses de perguntas. Pode também inserir perguntas ou validar questões de cujas classes esteja autorizado. Qualquer destas informações pode ser posteriormente consultada através das listagens disponíveis.

Vejamos um exemplo de como inserir uma classe no sistema.

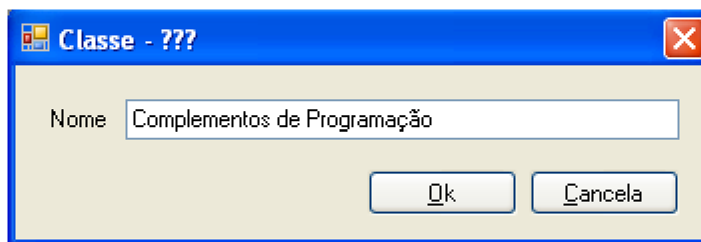


Ilustração 44 - Formulário para inserir classes no sistema

No caso da classe o único dado a inserir é a sua designação. A classe é posteriormente usada para classificar perguntas. Os jogos são passíveis de serem configurados no sentido de contemplarem apenas perguntas de certas classes. Os professores também têm associadas classes para que só possam aprovar certas perguntas (tipicamente as perguntas que leccionam).

As classes são ainda divididas em subclasses. A subclasse servirá como um filtro na configuração do jogo. As recomendações registam a bibliografia aconselhada para se ter sucesso nas respectivas questões.

Ilustração 45 - Formulário para inserir subclasses no sistema

Os estudantes podem inserir perguntas que ficarão sujeitas a aprovação ou rectificação, consultar questões e realizar jogos individualmente ou colectivamente.

Eis um formulário a preencher com os dados de uma pergunta.

Ilustração 46 - Formulário para inserir perguntas no sistema

Neste formulário indica-se a pergunta, uma ou mais respostas certas e respostas erradas (úteis essencialmente nas questões de escolha múltipla). A classe e subclasse permitem categorizar a questão. O campo “activa” indica que a pergunta está disponível para ser usada em jogos. O campo “aprova” só está disponível se a pergunta ainda não foi aprovada e se o utilizador corrente é um professor. Quando a informação for submetida, há várias regras de negócio que vão verificadas nomeadamente se a pergunta está preenchida e ainda não existe na base de dados, se há pelo menos uma resposta certa e se está classificada (tem classe e subclasse).

Após a inserção de uma pergunta por um estudante ela fica no estado “por aprovar”. Só depois de aprovada é que pode entrar nos jogos.

Para aprovar o professor entra no sistema e pede uma listagem de perguntas activas por aprovar de acordo com a Ilustração 47.

Ilustração 47 - Formulário para consulta de perguntas existentes na Base de Dados

Este formulário permite que se obtenha uma listagem de perguntas filtrada. No filtro podemos indicar opcionalmente o número da pergunta ou texto existente na pergunta.

Pode-se solicitar só perguntas activas, só de escolha múltipla, de um determinado nível de dificuldade, com uma determinada ordem ou só por aprovar. Pode-se também indicar a classe e subclasse das questões a visualizar. Quando se prime o botão pesquisa, obtêm-se na grelha as perguntas que respeitam o filtro indicado. Depois, se o utilizador fizer um duplo clique na pergunta acede ao formulário da pergunta. Se o utilizador for um professor pode fazer alterações ou validar a pergunta e, a partir daí a pergunta passa a estar disponível para os jogos. O formulário com a pergunta está exemplificado na Ilustração 48.

Ilustração 48 - Formulário de uma pergunta previamente inserida e passível de ser aprovada.

Anexo C - Utilizadores do Jogo

Os utilizadores do jogo estão descritos na Tabela 1.

Tabela 1 - Resumo dos Utilizadores

Nome	Descrição	Responsabilidades
Administrador	Pessoa responsável pela manutenção dos utilizadores e classes de perguntas.	Professor do estabelecimento de ensino que poderá realizar as seguintes acções: • Criar utilizadores; • Alterar os dados dos utilizadores, tais como, *email*, se está activo e qual o tipo de utilizador (Administrador, Estudante ou Professor). • Associar quais as classes de perguntas que um determinado docente pode revalidar; • Criar e alterar as classes de perguntas (pode por exemplo associar uma classe a uma disciplina).
Professor	Pessoa responsável por criar as subclasses das perguntas e validar as perguntas sugeridas pelos estudantes. Pode também inserir perguntas no sistema.	Professor do estabelecimento de ensino que poderá realizar as seguintes acções: • Criar subclasses às classes de perguntas que lhes estão associadas. As subclasses são normalmente usadas para distinguir as matérias leccionadas dentro de uma disciplina (classe); • Validar as perguntas propostas pelos alunos. Só após esta validação é que as perguntas podem ser usadas nos jogos. • O professor também pode inserir perguntas no sistema.

Nome	Descrição	Responsabilidades
Estudante	Pessoa que propõe perguntas e realiza jogos como forma complementar de estudo.	Poderá realizar as seguintes acções: • Propor perguntas para as diversas classes e subclasses existentes no sistema; • Realizar jogos individualmente ou em grupo com o objectivo de exercitar os seus conhecimentos e potenciar o sucesso nos exames.

Anexo D - Ambiente de Utilização do Jogo

O jogo deverá operar em computadores pessoais, num servidor aplicacional com funções de *relay* e num servidor de dados.

O ambiente de utilização é composto por vários utilizadores, tal como indica o diagrama de contexto da Ilustração 49.

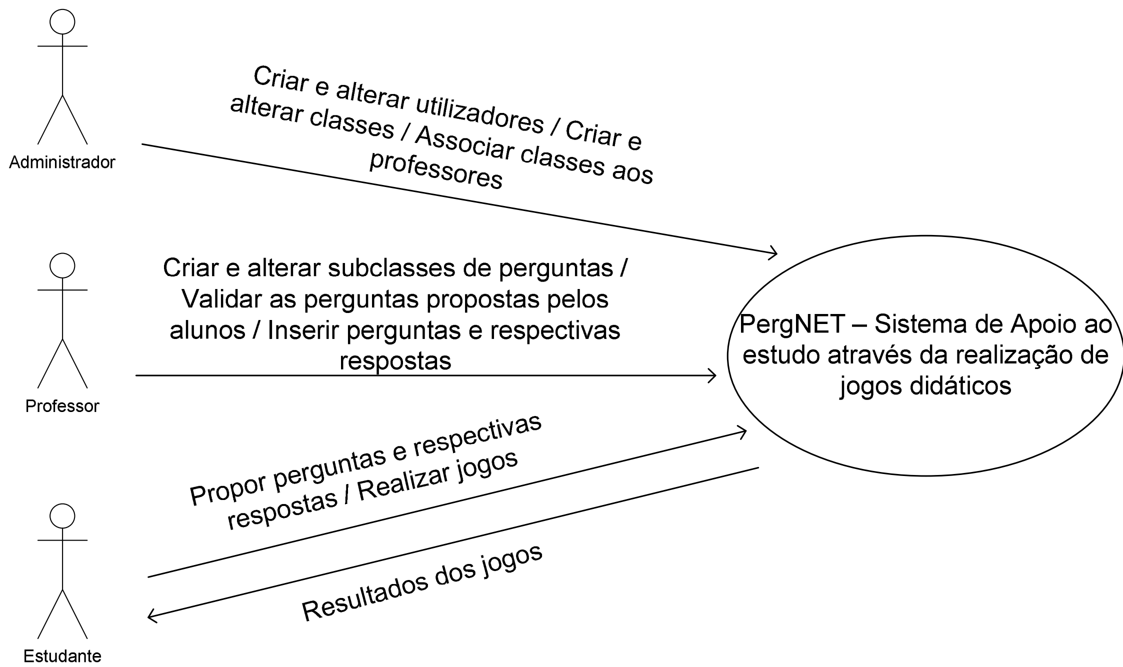


Ilustração 49 - Contexto de Utilização

Anexo E - Ficheiros que compõem a solução

A solução é composta por um conjunto de ficheiros que estão distribuídos pelas máquinas cliente e pelo Servidor. Os ficheiros em causa estão distribuídos da seguinte forma:

Tabela 2 - Ficheiros que compõem a solução

Ficheiro	Descrição	Cliente	Servidor
XTCPClient.dll	Sistema de comunicações para o cliente. Faz a gestão do *socket* TCP e respectivo contexto.	X	
XTCPServer.dll	Sistema de comunicações para o servidor. Faz a gestão do *socket* TCP e respectivo contexto.		X
XTCPShared.dll	Interfaces, utilitários e controladores de mensagem do sistema de comunicações.	X	X

Anexo F – Resumo dos serviços de iniciação e finalização

Tabela 3 - Serviços de iniciação e finalização solicitados pelo servidor da aplicação consumidora

Método	**Serviço**

`void StartTCPServer(`
`string serverIP,`
`int serverPort,`
`IMsgShower msgShower,`
`int keepAlivePeriod,`
`ISupplierId supplierId`
`)`
	Inicia o servidor de comunicações. É despoletado pelo servidor da aplicação consumidora. Como parâmetros recebe o IP e o Porto do servidor de comunicações, o visualizador de mensagens (existe para efeitos de *debugging* e, normalmente, é passado com "null"), o período a usar para o servidor verificar se os clientes estão vivos e um fornecedor de IDs para os futuros clientes.

`void StopTCPServer()`	Pára o servidor de comunicações.

Tabela 4 - Serviços de iniciação e finalização solicitados pelo cliente da aplicação consumidora

Método	Serviço

`void Start(`
`string serverIP,`
`int serverPort`
`)`
	Inicia o cliente de comunicações. É despoletado pelo cliente da aplicação consumidora. Como parâmetros recebe o IP e o Porto do servidor de comunicações.

`void Stop()`	Pára o cliente de comunicações.

Anexo G – Resumo dos serviços de notificação

Tabela 5 - Serviços de notificação (ocorrem em determinados eventos)

Evento	Delegate	Serviço
event NewElementHandler _newElementHandler;	delegate void NewElementHandler(string groupName, long newTcpClientId);	Aviso no cliente de que existe um novo elemento no grupo. O “groupName” indica qual é o grupo e o “newTcpClientId” indica qual é o ID do novo colaborador.
event QuitElementHandler _quitElementHandler;	delegate void QuitElementHandler(string groupName, long quitTcpClientId);	Aviso no cliente de que um elemento saiu do grupo. O “groupName” indica qual é o grupo e o “quitTcpClientId” indica qual é o ID do colaborador que saiu.
event DataArrivedFromEmitterHandler _dataArrivedFromEmitterHandler;	delegate void DataArrivedFromEmitterHandler(long clientId, IMsgCtrl msgCtrl, object data);	Aviso no servidor de que existe novos dados (que não necessitam de resposta) a reencaminhar para os clientes finais. O “clientId” indica o emissor dos dados. O “msgCtrl” é uma referência para o controlador da mensagem (a definição de controlador da mensagem pode ser vista no ponto 3.5.2). O objecto “data” contém os dados a enviar aos receptores.
event DataArrivedHandler _dataArrivedHandler	delegate void DataArrivedHandler(long clientId, IMsgCtrl msgCtrl, object data);	Aviso no cliente de que existe novos dados (que não necessitam de resposta). O “clientId” indica o receptor dos dados. O “msgCtrl” é uma referência para o controlador da mensagem. O objecto “data” contém os dados enviados ao receptor.

event DataWithAnswerArrivedFromEmitterHandler _dataWithAnswerArrivedFromEmitterHandler;	delegate void DataWithAnswerArrivedFromEmitterHandler(long clientId, IMsgCtrl msgCtrl, object data);	Aviso no servidor de que existe novos dados (que necessitam de resposta). O “clientId” indica o emissor dos dados. O “msgCtrl” é uma referência para o controlador da mensagem. O objecto “data” contém os dados a enviar aos receptores.
event DataWithAnswerArrivedHandler _dataWithAnswerArrivedHandler	delegate void DataWithAnswerArrivedHandler(long clientId, IMsgCtrl msgCtrl, object data);	Aviso no cliente de que existe novos dados (que necessitam de resposta). O “clientId” indica o receptor dos dados. O “msgCtrl” é uma referência para o controlador da mensagem. O objecto “data” contém os dados enviados ao receptor.
event DataWithAnswerArrivedToEmitterHandler _dataWithAnswerArrivedToEmitterHandler;	delegate void DataWithAnswerArrivedToEmitterHandler(long clientId, IMsgCtrl msgCtrl, object data);	Aviso no servidor de que existe novos dados para os quais já se obtiveram as respostas. O “clientId” indica o emissor dos dados. O “msgCtrl” é uma referência para o controlador da mensagem. O objecto “data” contém os dados a enviados aos receptores.

Anexo H – Resumo dos serviços que podem ser solicitados directamente pela aplicação

Tabela 6 – Serviços que podem ser solicitados pelo cliente da aplicação

Serviço	Descrição
long? Request_WhoAmI()	Devolve o ID do cliente.
void Request_SendData(object data, List<long> targetClientIds) ou void Request_SendData(object data, string groupName)	Envia avisos a uma lista de clientes ou a um grupo específico. Esta mensagem tem semântica assíncrona e não aguarda uma resposta. Esta mensagem ao chegar ao cliente provoca a geração de um evento (no cliente), permitindo que este execute código apropriado para este acontecimento (chegada de dados).
List<CAnswer> Request_SendDataAndReceiveAnswer(object data, List<long> targetClientIds) ou List<CAnswer> Request_SendDataAndReceiveAnswer(object data, string groupName)	Envia pedidos a uma lista de clientes ou a um grupo específico. Esta mensagem tem semântica síncrona e aguarda uma resposta. Esta mensagem quando chega ao cliente sinaliza um evento (no cliente), permitindo que este execute código apropriado para este acontecimento (chegada de dados). No *handler* deste evento coloca-se, normalmente, o tratamento dos dados, o processamento da resposta e a atribuição da resposta.
bool? Request_CreateGroup(string groupName)	Cria o grupo indicado e, caso consiga, regista-se no novo grupo.
bool? Request_AddMeToGroup(string groupName)	Adiciona o cliente corrente a um grupo já existente.
bool? Request_RemoveMeFromGroup(string groupName)	Remove o cliente corrente de um determinado grupo.
string[] Request_GetGroupsNames()	Devolve um *array* com os nomes dos grupos que existem.

`long[] Request_GetGroupClientIds(string groupName)`	Devolve um *array* de IDs de clientes que compõem um determinado grupo.
`string[] Request_GetLocalEndPoints()`	Devolve um *array* com todos os endereços locais de todos os elementos "vivos" e do servidor. Cada elemento do *array* está formatado de acordo com a expressão "ClienteId – LocalEndPoint".
`string Request_GetClientEndPoint(long clientId)`	Devolve o endereço IP/Porto com que o cliente foi registado no servidor (se entre o servidor e o cliente houver um NAT então o endereço em causa é o resultado do NAT).

Tabela 7 – Serviços que podem ser solicitados pelo servidor da aplicação

Serviço	**Descrição**
`void Request_HelloWorld(string message, long targetClientId)`	Envio de uma mensagem do servidor de comunicações a um determinado cliente.