PROTOCOLOS SEGUROS PARA JOGOS EM REDES PEER-TO-PEER

Bernardo de Melo Pacheco

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Geraldo Bonorino Xexéo

Rio de Janeiro
Setembro de 2013

PROTOCOLOS SEGUROS PARA JOGOS EM REDES PEER-TO-PEER

Bernardo de Melo Pacheco

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

____

Prof. Geraldo Bonorino Xexéo, D.Sc.

____

Prof. Jano Moreira de Souza, Ph.D.

____

Prof. José Antônio Moreira Xexéo, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2013

Pacheco, Bernardo de Melo

Protocolos Seguros para Jogos em Redes Peer-to-Peer/Bernardo de Melo Pacheco. – Rio de Janeiro: UFRJ/COPPE, 2013.

XIII, 119 p.: il.; 29, 7cm.

Orientador: Geraldo Bonorino Xexéo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2013.

Referências Bibliográficas: p. 106 – 111.

1. Rede peer-to-peer. 2. Protocolo criptográfico. 3. Jogo distribuído. I. Xexéo, Geraldo Bonorino. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha família, pelo eterno incentivo.

Agradecimentos

Agradeço a Deus, por me fortalecer nos meus desafios e por me ajudar em todos os momentos, principalmente nos mais difíceis.

Ao meu pai, José Felício Pacheco, pois sem você eu jamais estaria aqui. Sinto muito a sua falta e lamento por você ter me deixado durante o mestrado. Você, meu pai, trabalhou desde muito pequeno para sustentar a sua família, e abriu mão dos seus estudos. Porém, nunca conheci pessoa tão sábia. Se há um mestre aqui, este mestre é você.

À minha mãe, Verônica de Melo Pacheco, pelo seu amor materno. Seu apoio foi e ainda é fundamental para a minha criação e formação. Você é o meu porto seguro.

Aos meus irmãos, Humberto de Melo Pacheco, Eduardo de Melo Pacheco e Fernanda de Melo Pacheco, pela compreensão da minha ausência nos almoços e finais de semana.

À minha namorada e futura esposa Caroline Kurtemback. Sem você eu não teria conseguido. Você esteve sempre comigo, me apoiando, confortando e incentivando. Você deixou muitas coisas suas de lado pensando em mim. Muito obrigado.

À minha avó Rita de Mattos e amiga Cristina Mello pelo suporte familiar e também pelo delicioso feijão que me sustentou nos meus estudos.

Ao meu amigo e irmão Fenando Neves pelo apoio durante o mestrado e pela compreensão da minha ausência no último ano.

Ao meu orientador e professor Geraldo Bonorino Xexéo pela atenção, tempo, confiança e conhecimento gastos neste trabalho. Muito do que aprendi neste trabalho foi você que me mostrou o caminho. Saiba que aprendi muito, e sou muito grato.

Aos meus amigos da empresa DotLegend pela confiança e paciência: Bruno Goyanna, Carlos Eugênio, Daniel Gouvea, Fábio Freitas, Fábio Souza, Flávio Faria, Horácio França, João Pap, Júlio Brafman e Roberto Ferreira. Em especial, agradeço aos meus amigos Lúcio Paiva e Bruno França pela paciência que tiveram comigo durante este trabalho. Vocês me ensinaram muito, sempre com conselhos valiosos. Tenho certeza de que este trabalho enriqueceu por causa de vocês.

Aos meus amigos de graduação e mestrado Allan Girão, André Bakker, Arthur Siqueira, Diego Marins, Filipe Braida, Marcelo Justo, Pedro Ivo e Roque Pinel. Conquistamos muitas coisas juntos.

Ao professor Jano Moreira de Souza, pela oportunidade concedida para a realização do mestrado na linha de Banco de Dados e por aceitar participar da minha banca de defesa de mestrado.

Ao professor José Antônio Moreira Xexéo por participar da minha banca de defesa de mestrado e por contribuir com o seu conhecimento.

À CAPES, pelo apoio financeiro durante o mestrado.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROTOCOLOS SEGUROS PARA JOGOS EM REDES PEER-TO-PEER

Bernardo de Melo Pacheco

Setembro/2013

Orientador: Geraldo Bonorino Xexéo

Programa: Engenharia de Sistemas e Computação

Em um ambiente ponto-a-ponto, a segurança de uma aplicação está distribuída entre todos os participantes, ou seja, não há um servidor com o papel de parte confiável no sistema. Protocolos criptográficos são formas que determinam as regras de como cada participante se comunica, e que têm por objetivo garantir a justiça no sistema. *Mental Poker* e *Fair Coin Flipping* são problemas comuns tratados na literatura de criptografia com aplicações dentro e fora da área de jogos. *Mental Poker* trata dos problemas de gerenciar um baralho, garantindo, por exemplo, que as cartas são distribuídas com a mesma probabilidade entre os jogadores e que nenhum jogador consiga ver as cartas do outro. *Fair Coin Flipping* trata dos problemas de gerar um número aleatório, seja uma moeda ou um dado, garantindo que nenhum jogador consiga desbalancear o número gerado a seu favor.

No entanto, há uma carência de implementações práticas e de uso desses protocolos aplicados a jogos em um ambiente ponto-a-ponto. De fato existem estudos sendo realizados com protocolos criptográficos em conjunto com jogos, porém formalizados em um nível abstrato. Nesse contexto, este trabalho tem como objetivo estudar e propor uma biblioteca documentada que implementa um conjunto de protocolos criptográficos que suportam determinadas ações em jogos em um ambiente ponto-a-ponto, especificamente ações em jogos que utilizam a sorte por meio do jogo de dados ou cartas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SECURE PROTOCOLS TO GAMES IN PEER-TO-PEER NETWORK

Bernardo de Melo Pacheco

September/2013

Advisor: Geraldo Bonorino Xexéo

Department: Systems Engineering and Computer Science

In an peer-to-peer networking, the security of an application is distributed between all participants, i.e., there is no server as a trusted third party in the system. Cryptographic protocols are forms that determine the rules for how each participant communicates, and aim to ensure fairness in the system. Mental Poker and Fair Coin Flipping are common problems treated in the cryptographic literature with applications inside and outside the game area. Mental Poker deals with the problems of managing a deck, ensuring, for example, that the cards are distributed with equal probability among players and no player can see the cards of the other. The Fair Coin Flipping deals with the problems of generating a random number, a coin or a die, ensuring that no player can unbalance the number generated in your favor.

However, there is a lack of practical implementations and use of these protocols applied to games in an peer-to-peer networking. In fact there are studies being conducted with cryptographic protocols in conjunction with games, but formalized in an abstract level. In this context, this work aims to study and propose a library that implements a documented set of cryptographic protocols that support certain actions in games in an peer-to-peer networking, specific actions in games used by the luck of the dice or cards.

Sumário

Lista de Figuras **xi**

Lista de Tabelas **xiii**

1 Introdução **1**

1.1 Contexto e Motivação 1

1.2 Objetivos 3

1.3 Organização do Trabalho 3

2 Rede Ponto-a-Ponto e Protocolos Criptográficos **5**

2.1 Arquitetura ponto-a-ponto 5

2.1.1 Rede sobreposta 6

2.1.2 Camada de rede 7

2.1.3 Camada de aplicação 9

2.2 Teoria dos números 11

2.3 Criptografia e protocolos 14

2.3.1 Terminologia 14

2.3.2 Algoritmo simétrico 17

2.3.3 Algoritmo de chave pública 17

2.3.4 RSA 19

2.3.5 ElGamal 19

2.3.6 O que é um protocolo criptográfico? 21

2.3.7 *Bit Commitment* 22

2.3.8 *Secret Sharing* 25

2.3.9 *Fair Coin Flipping* 26

2.3.10 *Mental Poker* 30

3 Proposta de Biblioteca de Suporte a Ações em Jogos Ponto-a-Ponto **42**

3.1 Base de comunicação dos protocolos 42

3.2 Módulo *Dice Rolling* 44

3.2.1 Proposta de protocolo para n jogadores 44

3.2.2 Notação padrão de dados 47
3.2.3 Modelagem 48
3.2.4 Funcionamento interno 51
3.2.5 Como usar o módulo 52
3.3 Módulo *Mental Poker* 55
3.3.1 Proposta de extensão 56
3.3.2 Modelagem 61
3.3.3 Funcionamento interno 65
3.3.4 Como usar o módulo 70
3.4 Decisões de *desing* e padrões adotados 74

4 Implementação **77**
4.1 Estrutura do *peer* 77
4.2 Módulo *Dice Rolling* 79
4.3 Módulo *Mental Poker* 82

5 Experimento **85**
5.1 Configuração do ambiente 85
5.2 Experimento no módulo *Dice Rolling* 85
5.2.1 Cenário do experimento 86
5.2.2 Resultados 87
5.3 Experimento no módulo *Mental Poker* 92
5.3.1 Cenário do experimento 92
5.3.2 Resultados 94

6 Conclusão **102**
6.1 Considerações acerca do trabalho 102
6.2 Contribuições 103
6.3 Limitações e trabalhos futuros 105

Referências Bibliográficas **106**

A Resultados do experimento **112**
A.1 *Dice Rolling* 112
A.2 *Mental Poker* 115

Lista de Figuras

2.1 Comparação entre os tipos de arquitetura 6
2.2 Arquitetura de rede sobreposta ponto-a-ponto 7
2.3 Processo de criptografar e descriptografar 15
2.4 Processo de criptografar e descriptografar utilizando chave 16
2.5 Processo de criptografar e descriptografar utilizando duas chaves diferentes 17

3.1 Diagrama de classes que compõem a base da comunicação dos protocolos 43
3.2 Número de mensagens por número de jogadores 46
3.3 Diagrama de classes de projeto do módulo *Dice Rolling* 49
3.4 Diagrama de sequência das mensagens internas do módulo *Dice Rolling* na geração de dados 52
3.5 Diagrama de sequência de exemplo de uso do módulo *Dice Rolling* . . 55
3.6 Diagrama de classes de projeto do módulo *Mental Poker* 61
3.7 Diagrama de sequência de inicialização do módulo *Mental Poker* . . . 66
3.8 Diagrama de sequência de mensagens internas do módulo *Mental Poker* na geração de uma carta aberta 68
3.9 Diagrama de sequência de exemplo de uso do módulo *Mental Poker* . 74

4.1 Diagrama de classes do *peer* 78
4.2 Diagrama de sequência de conexão e recebimento de mensagem de um *peer* 78

5.1 Tempo médio de jogada no jogo *Dungeons & Dragons* 89
5.2 Tempo médio de jogada no jogo *Pathfinder* 89
5.3 Tempo médio de jogada no jogo *Vampire The Masquerade* 89
5.4 Tempo médio de criptografia por jogada no jogo *Dungeons & Dragons* 90
5.5 Tempo médio de criptografia por jogada no jogo *Pathfinder* 90
5.6 Tempo médio de criptografia por jogada no jogo *Vampire The Masquerade* 90
5.7 Tráfego médio por jogada no jogo *Dungeons & Dragons* 91
5.8 Tráfego médio por jogada no jogo *Pathfinder* 91

5.9 Tráfego médio por jogada no jogo *Vampire The Masquerade* 91
5.10 Tráfego médio por jogada no jogo de Maior Carta 95
5.11 Tráfego médio por jogada no jogo de Poker 95
5.12 Tráfego médio por jogada no jogo de Sete e Meio 96
5.13 Tráfego médio por jogada no jogo Escopa 96
5.14 Média de colisão por jogada no jogo de Maior Carta 96
5.15 Média de colisão por jogada no jogo de Poker 97
5.16 Média de colisão por jogada no jogo de Sete e Meio 97
5.17 Média de colisão por jogada no jogo Escopa 97
5.18 Tempo médio de criptografia por jogada no jogo de Maior Carta . . . 98
5.19 Tempo médio de criptografia por jogada no jogo de Poker 98
5.20 Tempo médio de criptografia por jogada no jogo de Sete e Meio . . . 98
5.21 Tempo médio de criptografia por jogada no jogo Escopa 99
5.22 Tempo médio para gerar uma carta aberta por jogada no jogo de Maior Carta 99
5.23 Tempo médio para gerar uma carta aberta na mesa por jogada no jogo de Poker 99
5.24 Tempo médio para gerar uma carta fechada por jogada no jogo de Poker 100
5.25 Tempo médio para gerar uma carta fechada por jogada no jogo de Sete e Meio 100
5.26 Tempo médio para gerar uma carta fechada por jogada no jogo Escopa 101

Lista de Tabelas

3.1 Comparação entre os tipos de notação de dados 48
3.2 Exemplo de mapeamento entre o número gerado pelo protocolo e as cartas do baralho 60

5.1 Notações de dado comuns 86
5.2 Jogos reais utilizados no experimento 87
5.3 Resultados das jogadas comuns 88
5.4 Aumento de performance no jogo de Maior Carta sem colisão 95
5.5 Aumento de performance no jogo Escopa no modo sem colisão com as rodadas anteriores 95

A.1 Resultados das métricas no jogo *Dungeons & Dragons* no modo normal 112
A.2 Resultados das métricas no jogo *Dungeons & Dragons* no modo em lote 113
A.3 Resultados das métricas no jogo *Pathfinder* no modo normal 113
A.4 Resultados das métricas no jogo *Pathfinder* no modo em lote 114
A.5 Resultados das métricas no jogo *Vampire The Masquerade* no modo normal 114
A.6 Resultados das métricas no jogo *Vampire The Masquerade* no modo em lote 115
A.7 Resultados das métricas no jogo Maior Carta 116
A.8 Resultados das métricas no jogo Pôquer 117
A.9 Resultados das métricas no jogo Sete e Meio 118
A.10 Resultados das métricas no jogo Escopa 119

Capítulo 1

Introdução

Este capítulo apresenta o contexto do trabalho e o que motivou esta pesquisa. São também apresentados os objetivos e a organização deste trabalho.

1.1 Contexto e Motivação

A segurança da informação nunca foi tão amplamente endereçada, utilizada, difundida e debatida entre os cidadãos comuns como nos dias de hoje. Por muito tempo, tudo relacionado à criptografia era exclusivamente de domínio militar. O paradigma mudou, e em vez de codificar e decodificar informações com os aliados e tentar decifrar as mensagens do inimigo, a criptografia atual vai além disso. Seja em uma troca de e-mails, em um acesso a uma página na internet ou em uma transação financeira, a criptografia está presente para garantir a segurança. De fato a criptografia é uma parte essencial dos sistemas de informação atuais. A criptografia evita fraudes no comércio eletrônico, impede que uma mensagem seja lida por pessoas não autorizadas, garante o anonimato e também permite provar sua identidade.

Para a grande maioria, a segurança da informação pode significar a segurança do projeto de um novo produto, uma nova estratégia de marketing da empresa, o plano para uma campanha política, os dados bancários, o extrato de compras do cartão de crédito daquele mês e até fotos pessoais da família. Manter uma carta em segredo, trancá-la em um cofre e esconder o cofre em alguma casa de qualquer cidade do mundo, não é segurança, e sim obscuridade. No entanto, se os detalhes do fucionamento interno do cofre são conhecidos, se outras cartas também são trancadas em outros cofres, e se qualquer pessoa do mundo estudar como o cofre pode ser aberto, e mesmo assim ninguém conseguir ler a carta, isso é segurança (SCHNEIER, 1996).

A segurança é um requisito ainda mais evidente e necessário em um ambiente

ponto-a-ponto[1] com a participação de dois ou mais participantes. Ao contrário da arquitetura cliente-servidor[2] onde uma terceira parte confiável[3] gerencia e garante a confiabilidade do sistema, em um sistema ponto-a-ponto geralmente não existe essa âncora que atua como entidade confiável. Dessa forma, em um sistema ponto-a-ponto, a confiabilidade do sistema é distribuída entre todos os participantes que seguem as regras de um protocolo criptográfico. Por exemplo, a tarefa de dividir um pedaço de bolo ao meio entre duas pessoas caracteriza bem a essência de um protocolo criptográfico. Existem várias formas de partir um bolo ao meio. A primeira solução é encontrar uma outra pessoa para partir o bolo. O problema é que é necessário a participação de outra pessoa. Na segunda solução uma das duas pessoas parte o bolo e a outra pode queixar-se com a autoridade caso os pedaços sejam diferentes. Ainda assim outra pessoa deve participar. Por fim, como solução do problema, uma pessoa parte o bolo e a outra escolhe o pedaço que desejar. Essa solução não requer a participação de nenhuma outra pessoa e ainda garante que o bolo será repartido igualmente. Portanto, o objetivo de um protocolo criptográfico é resolver um problema entre dois ou mais participantes, sejam eles amigos, inimigos, desconhecidos, honestos ou desonestos, de forma que quem trapaceia é detectado e punido. Além disso, ninguém aprende mais do que o outro participante ou do que o protocolo permite.

Um protocolo criptográfico pode ser constituído por outros protocolos criptográficos que realizam uma tarefa específica. As aplicações de um protocolo criptográfico são diversas, incluindo o processo de troca e definição da chave criptográfica utilizada em uma sessão de comunicação; o processo de autenticação que garante que ninguém assuma a identidade alheia; o particionamento de um segredo entre n indivíduos sendo necessário um número mínimo de indivíduos para reconstruir o segredo; o comprometimento de um indivíduo com uma informação que não pode ser alterada e que será revelada somente no futuro; o processo de provar que alguém conhece um segredo sem revelar este segredo; a assinatura de um documento sem revelar o conteúdo do mesmo; a condução de uma eleição onde é possível obter o resultado final sem que haja trapaças e que mantém a identidade dos indivíduos; a operação de jogar par ou ímpar, moeda ou dados via telefone, e-mail ou por qualquer outro meio de mensagens e a operação de jogar cartas sem um baralho físico.

Em um ambiente cliente-servidor, um jogo que tem elementos de sorte e que usa dados e cartas mantém toda a segurança e justiça no servidor. O servidor atua como um juiz em que todos os jogadores confiam, o juiz que gera, aleatoriamente, o número do dado e a carta do baralho. No entanto, em um ambiente ponto-a-

[1]Tradução do termo em inglês *peer-to-peer*

[2]Tradução do termo em inglês *client-server*

[3]Tradução do termo em inglês *trusted third party*

ponto, tal juiz não existe. Para gerar um dado ou uma carta do baralho, deve-se garantir que nenhum jogador possa levar o resultado a ser desbalanceado a seu favor. *Mental Poker* e *Fair Coin Flipping* são problemas comuns tratados na literatura de criptografia com aplicações dentro e fora da área de jogos. O *Mental Poker* trata dos problemas de gerenciar um baralho, garantindo, por exemplo, que as cartas são distribuídas com a mesma probabilidade entre os jogadores e que nenhum jogador consiga ver as cartas de outro jogador. O *Fair Coin Flipping* trata dos problemas de gerar um número aleatório, seja uma moeda ou um dado, garantindo que nenhum jogador consiga desbalancear o número gerado a seu favor.

Apesar das várias aplicações em que um protocolo criptográfico pode ser utilizado, há uma carência de implementações práticas e de uso desses protocolos aplicados a jogos em um ambiente ponto-a-ponto. De fato existem estudos sendo realizados com protocolos criptográficos em conjunto com jogos, porém formalizados em um nível abstrato. Portanto, este trabalho tem como objetivo estudar e propor uma biblioteca documentada que implementa um conjunto de protocolos criptográficos que suportam determinadas ações em jogos em um ambiente ponto-a-ponto, especificamente ações em jogos que utilizam a sorte por meio de dados ou cartas. Este trabalho não é uma implementação de um jogo, mas sim um estudo aplicado de criptografia por meio de protocolos criptográficos em jogos.

1.2 Objetivos

Os objetivos deste trabalho são:

- Realizar um levantamento do referencial teórico relacionado a protocolos criptográficos aplicados a jogos com o objetivo de encontrar oportunidades de implementação;

- Propor, modelar e implementar uma biblioteca de suporte a ações específicas em jogos em um ambiente ponto-a-ponto, sendo estas ações relacionadas à sorte por meio de dados e cartas. Além disso, propor e implementar extensões de protocolos criptográficos existentes;

- Analisar a complexidade e o desempenho das operações da biblioteca proposta, além de evidenciar os resultados obtidos por meio da condução de experimentos variando o número de jogadores participantes.

1.3 Organização do Trabalho

Esta dissertação está organizada da seguinte maneira:

O capítulo 2 discute os conceitos referentes à fundamentação teórica do trabalho. São abordados os conceitos de rede ponto-a-ponto, teoria dos números, criptografia e protocolos criptográficos.

No capítulo 3 será apresentada a proposta de biblioteca de suporte a ações em jogos num ambiente ponto-a-ponto.

No capítulo 4 será descrita a implementação da biblioteca proposta.

No capítulo 5 será apresentado o experimento e os resultados que avaliam as operações da biblioteca.

Por fim, são apresentadas a conclusão e as referências bibliográficas deste trabalho.

Capítulo 2

Rede Ponto-a-Ponto e Protocolos Criptográficos

Para que seja possível o entendimento das técnicas e tecnologias utilizadas, este capítulo apresenta os conceitos teóricos de maior relevância. As seções tratam os seguintes assuntos: arquitetura ponto-a-ponto, onde será descrita essa arquitetura e também o contexto onde este trabalho se encaixa; seção de teoria dos números que apresenta a base matemática deste trabalho; seção de criptografia que descreve os conceitos, técnicas e os protocolos criptográficos que são a base da biblioteca que será proposta neste trabalho.

2.1 Arquitetura ponto-a-ponto

A arquitetura ponto-a-ponto ganhou notoriedade no final da década de 90 principalmente pelo serviço de compartilhamento de arquivos NAPSTER (2012), já que o serviço permitia às pessoas compartilhar qualquer música sem a necessidade de pagar por elas. Mais ainda, esse tipo de arquitetura ganhou popularidade por ser um mecanismo onde os usuários compartilhavam arquivos sem a necessidade de servidores centralizados. Trata-se, em outras palavras, de uma mudança de paradigma à usual arquitetura cliente-servidor. Esse novo paradigma provê uma poderosa plataforma para a construção de serviços descentralizados, como armazenamento na rede, distribuição de conteúdo, cache para a web, busca e indexação mais ricas no nível da aplicação.

Para ORAM (2001) a arquitetura ponto-a-ponto é "uma classe de aplicações que tira proveito dos recursos – armazenamento, ciclos, conteúdo, presença humana – disponíveis nas extremidades da internet. Porque acessando estes recursos descentralizados significa operar em um ambiente de conectividade instável e de endereço IP imprevisível; os nós da rede devem operar fora do DNS e ter uma autonomia

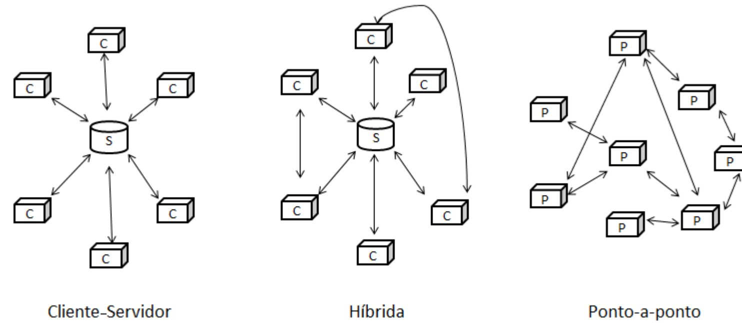

Figura 2.1: Comparação entre os tipos de arquitetura (adaptado de SCHOLLMEIER (2001); STEINMETZ e WEHRLE (2005))

significativa a partir dos servidores centrais." Em outras palavras, a arquitetura ponto-a-ponto é um sistema distribuído na camada da aplicação, onde os nós podem se comunicar entre si por meio de um protocolo de roteamento na camada de rede.

Para diferenciar de fato uma arquitetura ponto-a-ponto da clássica arquitetura cliente-servidor, a arquitetura ponto-a-ponto pode ser descrita como puramente ponto-a-ponto[1] ou ponto-a-ponto híbrida[2](SCHOLLMEIER, 2001; STEINMETZ e WEHRLE, 2005). A primeira assume que os serviços providos pela rede ponto-a-ponto não serão impactados com a saída de um nó participante. Já a segunda assume a participação de um nó centralizador como parte necessária para prover os serviços da rede. Dessa forma, caso esse nó saia da rede, os serviços podem ser parcialmente prejudicados. Contudo, a interação entre os outros nós ainda permite que o sistema continue ativo. Na arquitetura cliente-servidor, somente o servidor fornece conteúdo e provê serviços para os demais clientes da rede. Caso o servidor falhe, todo o sistema falha.

2.1.1 Rede sobreposta

Uma rede sobreposta[3] é um conjunto de nós e ligações lógicas construídas em cima de uma rede existente com a finalidade de implementar serviços que não estão disponíveis (AL-OQILY e KARMOUCH, 2007).

Uma rede sobreposta ponto-a-ponto procura fornecer uma série de características, tais como seleção de nós que estão próximos, armazenamento redundante, busca e localização eficiente de dados, garantia de persistência de dados, autenticação e anonimato (LUA *et al.*, 2005). Segundo LUA *et al.* (2005), a arquitetura de rede sobreposta ponto-a-ponto pode ser segmentada em 5 camadas:

[1]Tradução do termo inglês *pure P2P*
[2]Tradução do termo inglês *hybrid P2P*
[3]Tradução do termo inglês *overlay network*

- Camada de rede de comunicação: descreve as características de comunicação entre dispositivos ligados de uma forma *ad-hoc*;
- Camada de gestão de sobreposição de nós: abrange o gerenciamento dos nós, como descoberta de nós e algoritmos de roteamento;
- Camada de gerenciamento de recursos: abrange segurança, confiabilidade e tolerância a falhas a fim de manter a solidez do sistema;
- Camada de serviços: serviços que suportam a camada de aplicação, como escalonamento de tarefas e gerenciamento de arquivos;
- Camada de aplicação: aplicações e ferramentas que são implementadas com funcionalidades específicas.

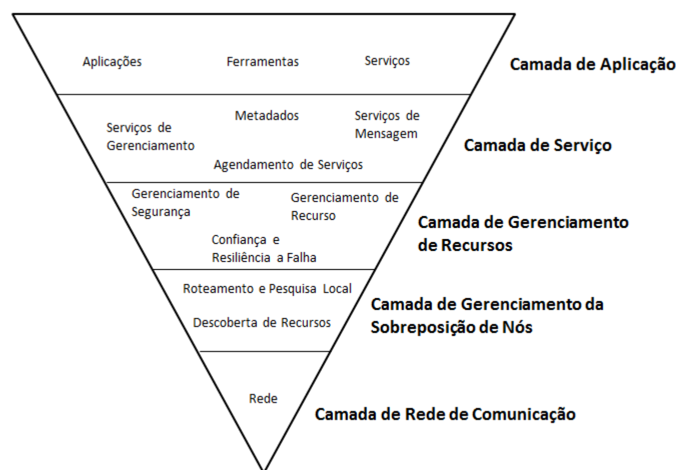

Figura 2.2: Arquitetura de rede sobreposta ponto-a-ponto (adaptado de LUA *et al.* (2005))

De fato o trabalho desenvolvido nesta dissertação faz parte da camada de aplicação, pois propõe e desenvolve uma biblioteca que serve como ferramenta para a construção de jogos em um ambiente ponto-a-ponto.

2.1.2 Camada de rede

A partir da perspectiva da camada de rede, ROUSSOPOULOS *et al.* (2003) acredita que um sistema ponto-a-ponto deve satisfazer os seguintes requisitos:

- Auto-organização[4]: Por meio de um processo de descoberta, os nós se organizam e se comunicam. Não há diretório global de nós ou recursos;

[4]Tradução do termo em inglês *self-organization*

- Comunicação simétrica[5]: Os nós são considerados iguais. Um nó pode tanto requisitar ou fornecer serviços;
- Controle distribuído[6]: O nó determina o seu nível de participação. Não há um controlador central que diz qual é o comportamento de um nó individual.

WANG e LI (2003) argumentam que há três importantes componentes em uma rede ponto-a-ponto: descoberta de vizinho[7], protocolo de localização[8] e protocolo de roteamento[9]. Um nó que acabou de entrar no sistema deve obter algumas informações básicas, tais como os nós vizinhos. Além disso, o novo nó também deve publicar as informações de quais objetos ele possui. O nó pode consultar por objetos que ele deseja. Neste momento, o protocolo de localização descobrirá qual o nó que contém o objeto requisitado, enquanto o protocolo de roteamento encaminha a mensagem de consulta para o nó que contém o objeto. A partir do IP do nó que contém o objeto consultado, o nó pode pegar o objeto desejado. Por fim o nó anuncia a sua saída da rede.

Protocolos de localização e roteamento como CAN (RATNASAMY *et al.*, 2001), Chord (STOICA *et al.*, 2001), Pastry (ROWSTRON e DRUSCHEL, 2001) e Tapestry (ZHAO *et al.*, 2001) são uma camada de rede estruturada e auto-organizável que têm como principais objetivos garantir requisitos como escalabilidade, eficiência, tolerância a falhas e balanceamento de carga efetivo. Esses protocolos permitem à aplicação localizar qualquer objeto em uma probabilidade delimitada, com um pequeno número de roteamento na rede, enquanto que cada nó armazena uma pequena tabela de roteamento com poucas entradas (WALLACH, 2002).

WALLACH (2002) discute questões de segurança que ocorrem no escopo dos protocolos de localização e roteamento. Segundo ele, a arquitetura deve ser robusta o suficiente para evitar que alguns nós agindo em conjunto ataquem outros nós. Nós maliciosos podem responder a uma requisição com dados falsos, retornando rotas falsas a fim de particionar a rede. Esse nós podem ter outros objetivos como a coleta e análise de informações num sistema que tenta prover comunicação anônima, além da censura a sistemas que tentam prover alta disponibilidade de informação.

CASTRO *et al.* (2002) reflete que uma rede ponto-a-ponto deve ser capaz de suportar um ambiente aberto onde os participantes têm interesses conflitantes entre si. Portanto, em uma rede de larga escala, é factível assumir a presença de nós maliciosos que podem corromper mensagens ou informar rotas erradas. Tais nós ainda podem assumir a identidade de outros nós para corromper ou deletar objetos

[5]Tradução do termo em inglês *symmetric communication*
[6]Tradução do termo em inglês *distributed control*
[7]Tradução do termo em inglês *neighbor finding*
[8]Tradução do termo em inglês *location protocol*
[9]Tradução do termo em inglês *routing protocol*

em nome do nó verdadeiro.

Portanto, é notável que a segurança é um requisito importante tanto na camada mais baixa, ou seja, na camada de rede quanto na camada de aplicação descrita a seguir.

2.1.3 Camada de aplicação

STEINMETZ e WEHRLE (2005) argumentam que a tradicional arquitetura cliente-servidor requer um esforço muito maior para acompanhar a crescente expansão da internet, assim como atender requisitos de novas aplicações e serviços que surgem com essa expansão. Segundo eles, esses requisitos são:

- Escalabilidade[10]: pode ser entendida como a capacidade de crescimento da aplicação ou do serviço em termos computacionais à medida que o número de usuários cresce.
- Segurança[11]: Serviços devem sempre estar disponíveis mesmo com ataques maliciosos. Além disso, o anonimato deve ser preservado quando requerido e a censura deve ser repudiada.
- Flexibilidade e qualidade de serviço[12]: Serviços devem ser flexíveis o suficiente para integrar novas tecnologias a fim de prover serviços de mais qualidade.

A arquitetura cliente-servidor, por sua natureza centralizadora, apresenta gargalos na disposição de recursos computacionais, além de apresentar um ponto único de falha. STEINMETZ e WEHRLE (2005) acreditam que a arquitetura ponto-a-ponto apresenta uma mudança de paradigma suficiente para endereçar esses requisitos.

Para KUBIATOWICZ (2003) uma aplicação ponto-a-ponto deve apresentar os seguintes requisitos:

- Disponibilidade[13]: Informação deve estar disponível 24 horas ao dia, sete dias da semana;
- Durabilidade[14]: Informação que entra no sistema deve permanecer no sistema;
- Controle de acesso[15]: A informação é protegida. Quem não é autorizado não pode ler nem alterar a informação;

[10]Tradução do termo em inglês *scalability*
[11]Tradução dos termos em inglês *security, reliability*
[12]Tradução dos termos em inglês *flexibility, quality of service*
[13]Tradução do termo em inglês *availability*
[14]Tradução do termo em inglês *durability*
[15]Tradução do termo em inglês *access control*

- Autenticidade[16]: Um documento requisitado não pode ser substituído por outro documento forjado;
- Recuperação de falha de serviço[17]: Deve ser difícil comprometer a disponibilidade do sistema.

Além desses requisitos, KUBIATOWICZ (2003) lista outros objetivos que podem ser endereçados de forma não obrigatória:

- Escalabilidade[18]: O sistema deve funcionar com milhares, milhões ou até bilhões de usuários;
- Anonimato[19]: Deve ser impossível ou muito difícil para um observador saber quem produziu um documento;
- Negação[20]: Usuários podem negar o conhecimento dos dados armazenados em sua máquina;
- Resistência à censura[21]: Ninguém pode censurar qualquer tipo de informação uma vez que esta seja inserida no sistema.

Os requisitos apresentados acima são de fato requisitos relacionados à segurança da aplicação. Como será descrito neste trabalho, essa segurança é o foco dos protocolos criptográficos.

Para RISSON e MOORS (2007) a pesquisa na área de arquitetura ponto-a-ponto pode ser dividida em quatro linhas de pesquisa: busca, segurança, armazenamento e aplicações. A linha de busca trata da indexação da informação, bem como as consultas para recuperar essa informação. A linha de armazenamento está relacionada com a troca, consistência e replicação da informação entre os nós. A linha de pesquisa que trata da segurança procura dispor essa informação baseada na reputação dos nós, assim como garantir que a troca de informações entre os nós seja justa, ou seja, que não tenha qualquer tipo de fraude. Já a linha de aplicação procura juntar o conhecimento das outras três linhas de pesquisa a fim de criar aplicações e serviços sobre essa rede, como sistema de arquivos distribuídos e serviços de *streaming* por exemplo.

Este trabalho está inserido nas linhas de pesquisa de segurança e aplicação. Serão vistas no decorrer deste trabalho as particularidades do projeto de jogos (aplicação) de natureza distribuída onde é possível a troca de informações entre os jogadores

[16] Tradução do termo em inglês *authenticity*
[17] Tradução do termo em inglês *denial-of-Service - DoS - resilience*
[18] Tradução do termo em inglês *scalability*
[19] Tradução do termo em inglês *anonymity*
[20] Tradução do termo em inglês *deniability*
[21] Tradução do termo em inglês *resistance to censorship*

de uma forma justa e segura (segurança) a fim de evitar qualquer tipo de trapaça e vantagem de um dos jogadores.

2.2 Teoria dos números

Este trabalho descreve e implementa protocolos criptográficos. Para compreendê-los é necessário entendimento em conceitos da área de teoria dos números. A teoria dos números é a parte da matemática que estuda as propriedades dos números inteiros, e de fato é uma das áreas mais antigas da matemática. O objetivo nessa seção é citar alguns desses conceitos. Para uma abordagem mais aprofundada do assunto, pode-se consultar IRELAND e ROSEN (1990), NIVEN *et al.* (2008) e HARDY e WRIGHT (1979). Outro excelente material é o livro *Números Inteiros e Criptografia RSA*(COUTINHO, 2003) utilizado no curso de graduação de Ciência da Computação na UFRJ.

Números inteiros

Os números ..., -3, -2, -1, 0, 1, 2,... são chamados de *números inteiros* e pertencem ao conjunto dos inteiros $\mathbb{Z}$. As letras *a, b,..., n, p,..., x, y,...* geralmente representam um número inteiro que pode estar com uma restrição do tipo número positivo ou número negativo. Um inteiro a é divisível por outro inteiro b, sem ser o zero, caso exista um inteiro c onde $a = bc$. O fato de a ser divisível por b é representado como $b \mid a$.

Números primos

No conjunto de números inteiros, há um subgrupo que desempenha um papel fundamental em criptografia, o grupo dos *números primos*. Um número p é primo se (i) $p > 1$ e (ii) p não tem divisor exceto 1 e o próprio p. Caso constrário, o número é chamado composto. O número inteiro 1 não é primo nem composto. A lista inicial de número primos é 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 41, 43, 47, 53, 59, 61... e de fato existem infinitos números primos. O Teorema Fundamental da Aritmética sustenta que cada inteiro positivo, exceto 1, pode ser escrito como um produto de primos. Um número n composto pode ser escrito como $n = p_1 p_2 ... p_k$. Vale destacar que decompor um número inteiro em primos pode ser um processo muito lento.

Máximo divisor comum

Entre dois inteiros a e b, o *máximo divisor comum* é o maior inteiro positivo d que é divisor de a e também é divisor de b. Se *mdc* é o máximo divisor comum entre a e b, escreve-se *mdc(a, b)*. Se *mdc(a, b)* $= 1$, a e b são *primos entre si* ou *co-primos.*

Aritmética modular

Na aritmética modular, um conjunto de inteiros estão contidos em um valor limite chamado módulo. Dois inteiros a e b são *congruentes* módulo n se a - b é um múltiplo de n. Basicamente, $a \equiv b \pmod{n}$ se $a = b + kn$ para algum inteiro k. O sinal $\equiv$ significa *congruência*. O número inteiro b é o resto da divisão de a por n. O número inteiro b também é chamado de *resíduo* de a módulo n. O conjunto de inteiros de 0 até n - 1 forma o conjunto completo de resíduos módulo n, o que significa que para cada inteiro a, seu resíduo módulo n é um número entre 0 e n - 1.

A relação do conjunto $\mathbb{Z}$ com a congruência módulo n forma um conjunto que é muito utilizado em criptografia. Este conjunto tem a notação $\mathbb{Z}_n$, e um nome de *conjunto dos inteiros módulo n*. Nesse conjunto, pode-se realizar operações aritméticas como soma, subtração e multiplicação. Por exemplo, 6 + 2 em $\mathbb{Z}_5$ é igual a 3, já que $6 + 2 = 8 \equiv 3 \pmod{5}$.

Elemento inverso

O *elemento inverso*, ou a inversão de um número inteiro a módulo n em $\mathbb{Z}_n$ é um número inteiro x que multiplicado por a tem congruência 1 com n. Por exemplo, o inverso de 4 módulo 7 pode ser escrito como $4 * \mathrm{x} \equiv 1 \pmod{7}$, com $x = 2$. O problema em geral é achar um x tal que $1 = (a * x) \bmod n$. A mesma sentença também é escrita como $a^{-1} \equiv x \pmod{n}$. Achar o inverso de um número pode ser uma tarefa muito difícil de resolver, podendo ter uma solução ou não. Em geral, $a^{-1} \equiv x \pmod{n}$ tem uma solução em $\mathbb{Z}_n$ se a e n são co-primos. Caso contrário, $a^{-1} \equiv x \pmod{n}$ não tem solução. Se n é um número primo, então cada número de 1 até n-1 é co-primo com n e tem um número inverso módulo n nesse intervalo. A função φ de *Euler* ou *função totiente* de um número n, escrita como $\varphi(n)$, é o número de inteiros positivos menores que n que são co-primos de n. Se n é primo, então $\varphi(n) = n$ - 1. Se $n = pq$, onde p e q são primos, então $\varphi(n) = (p - 1)(q - 1)$. Essa sentença é importante para a criptografia de chave pública RSA, já que a segurança do método está na dificuldade de fatorar um número inteiro n nos números inteiros primos p e q, onde $n = pq$. Usando a mesma função totiente, é possível calcular o elemento inverso x de um número inteiro a, onde $x = a^{\varphi(n)-1} \bmod n$. Por exemplo, para determinar o inverso do número 5 módulo 7, calcula-se $\varphi(7) = 7 - 1 = 6$, então o inverso é $5^{6-1} \bmod 7 = 5^5 \bmod 7 = 3$.

Resíduo quadrático

Sendo p um número primo, e a um número inteiro maior que 0 e menor que p, a é um *resíduo quadrático* módulo p se $x^2 \equiv a \pmod{p}$ para algum x. Por exemplo, $1^2 \equiv 1 \pmod{7}$, $2^2 \equiv 4 \pmod{7}$, $3^2 \equiv 2 \pmod{7}$. Os números 1, 2 e 4 formam o

conjunto de resíduos quadráticos em $\mathbb{Z}_7$. Os outros elementos 3, 5 e 6 são *resíduos não quadráticos* de $\mathbb{Z}_7$. De fato achar um número a elevado ao quadrado que é congruente a x módulo p pode ser tão difícil quanto achar os fatores primos p e q no algoritmo RSA.

Grupos

Um *grupo* é uma estrutura constituída de um *conjunto* G e uma *operação* * definida neste conjunto. A operação é uma regra que a cada dois elementos a, $b \in$ G associa um terceiro elemento $a * b$ que também está em G. Como exemplo de conjunto e operação, há o conjunto dos inteiros e a soma, os inteiros e o produto, os racionais e o produto, entre outros.

Nem todo conjunto e operação formam um grupo. Deve-se satisfazer as seguintes leis de formação para ser considerado um grupo:

1. **Associatividade**: dados a, b, $c \in$ G temos que $a * (b * c) = (a * b) * c$;

2. **Elemento neutro**: existe um elemento $e \in$ G tal que para todo $a \in$ G temos $a * e = e *$ a = a;

3. **Elemento inverso**: dado um elemento $a \in$ G, existe um elemento $a^{'} \in$ G que é o inverso de a tal que $a * a^{'} = a^{'} * a = e$.

Um grupo G não necessariamente precisa ter uma operação comutativa, ou seja, $a * b = b * a$ para quaisquer elementos a, $b \in$ G. Um grupo cuja operação é comutativa é chamado de *abeliano*. Como exemplo, o conjunto dos inteiros $\mathbb{Z}$ com a operação de adição forma um grupo abeliano. O número de elementos de um grupo é a sua *ordem*. De fato a ordem do grupo dos inteiros $\mathbb{Z}$ tem ordem infinita. No entanto, para o grupo $\mathbb{Z}_n$ dos inteiros módulo n com a operação de soma, a ordem do grupo é *finita* e igual a n.

Supondo que G é um grupo finito com uma operação *, e a é um elemento de G. Pode-se dizer que $a^k = a * a * a * ... * a$ (k vezes). Dessa forma, G = $\{e, a, a^2, a^3, ...\}$, e existe um elemento qualquer $a \in$ G com um inteiro positivo k tal que $a^k = e$. Pode-se dizer que se um grupo G é igual ao conjunto de potências de um elemento a, o elemento a é denominado um *gerador* do grupo G. O menor inteiro positivo k tal que $a^k = e$ é a *ordem* de a. Dessa forma, se G tem como gerador o elemento a, então a ordem de G é igual a ordem de a. Um grupo que admite um elemento gerador é chamado de *cíclico*. Por exemplo, o grupo $\mathbb{Z}_n$ é um importante grupo que é abeliano e cíclico.

Logaritmo discreto

Exponenciação modular é uma expressão do tipo $a^x \bmod n$ e pode ser facilmente calculada. No entanto, o problema inverso da exponenciação modular é um problema difícil de calcular, e é chamado de *logaritmo discreto*. Achar o logaritmo discreto é encontrar um número inteiro x onde $a^x \equiv b \pmod{n}$. Por exemplo, se $3^x \equiv 15 \pmod{17}$, então $x = 6$. Nem todo logaritmo discreto tem solução. A segurança de muitos algoritmos de chave pública é baseada no problema de encontrar o logaritmo discreto, com a complexidade de achar a solução semelhante ao problema de fatorar um número inteiro grande. O problema torna-se ainda mais difícil de resolver com números de 1024 *bits*.

2.3 Criptografia e protocolos

Nesta subseção serão apresentados os fundamentos da criptografia utilizada nesse trabalho, abordando a terminologia, a ideia principal de criptografar e descriptografar uma informação e os tipos de algoritmos criptográficos. Também será descrito o que é um protocolo criptográfico, além de descrever passo a passo os protocolos criptográficos de interesse neste trabalho.

2.3.1 Terminologia

Segundo MENEZES *et al.* (1996), Criptografia é o estudo de técnicas matemáticas relativas aos aspectos de segurança da informação, tais como:

- Confidencialidade[22]: é um serviço que tem por objetivo manter o acesso a uma informação somente a quem tem autorização de acesso;

- Integridade de dados[23]: é um serviço que endereça a alteração de dados por um intruso, ou seja, por alguém não autorizado. Para garantir a integridade dos dados, a parte receptora da mensagem deve ser capaz de detectar que a mensagem original foi alterada durante o envio;

- Autenticação[24]: é um serviço relacionado à identificação. Dois participantes em uma comunicação devem se identificar. Deve ser possível que receptor da mensagem se assegure de que não está recebendo uma mensagem de um intruso;

[22]Tradução do termo em inglês *confidentiality*
[23]Tradução do termo em inglês *data integrity*
[24]Tradução do termo em inglês *authentication*

- Não Repúdio[25]: um remente não deve ser capaz de negar mais tarde que enviou uma determinada mensagem.

De modo geral, em criptografia temos um participante que deseja enviar uma mensagem para outro participante. O participante que envia a mensagem deseja que a mesma seja transmitida de forma segura de forma que nenhum outro observador possa ler essa mensagem. Essa mensagem[26] pode ser denotada por M, e pode representar um conjunto de *bits*, um arquivo texto, uma imagem ou até mesmo um vídeo.

Criptografar[27] uma mensagem significa transformar a informação contida nessa mensagem a fim de impossibilitar a leitura de todos exceto o destinatário da mensagem. Por outro lado, descriptografar[28] uma mensagem significa transformar uma mensagem criptografada[29] a fim de obter a mensagem original. Uma mensagem criptografada é denotada por C.

O processo de criptografar uma mensagem pode ser descrito matematicamente como:

- $E(M) = C$, onde E é uma função que criptografa a mensagem M.

O processo de descriptografar pode ser descrito matematicamente como:

- $D(C) = M$, onde D é uma função que descriptografa a mensagem M.

Descriptografar uma mensagem criptografada resulta na própria mensagem original. Dessa forma, chegamos a seguinte igualdade:

- $D(E(M)) = M$

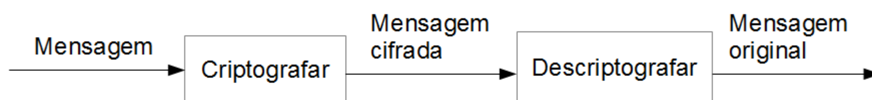

Figura 2.3: Processo de criptografar e descriptografar (adaptado de SCHNEIER (1996, p 1))

Um algoritmo criptográfico[30], é uma função matemática usada para criptografar e descriptografar uma mensagem. Comumente existe uma função para criptografar e outra função para descriptografar. Ao contrário dos antigos algoritmos que eram

[25] Tradução do termo em inglês *non-repudiation*
[26] Tradução dos termos em inglês *message*, *plaintext*
[27] Tradução do termo em inglês *encryption*
[28] Tradução do termo em inglês *decryption*
[29] Tradução do termo em inglês *ciphertext*
[30] Tradução do termo em inglês *cipher*

seguros por manter os detalhes do algoritmo em segredo, os modernos algoritmos expõem os seus detalhes de implementação, porém usam uma chave secreta para criptografar e descriptografar uma mensagem. Essa chave[31] é denotada por K. Toda segurança desse algoritmo reside na chave, e não de como o algoritmo funciona. Se um observador conhecer os detalhes do algoritmo, mas não conhecer a chave usada na criptografia da mensagem, a mensagem não poderá ser lida.

Temos, então, a seguinte notação usando uma chave para criptografar e descriptografar uma mensagem:

$$E_k(M) = C$$

$$D_k(C) = M$$

$$D_k(E_k(M)) = M$$

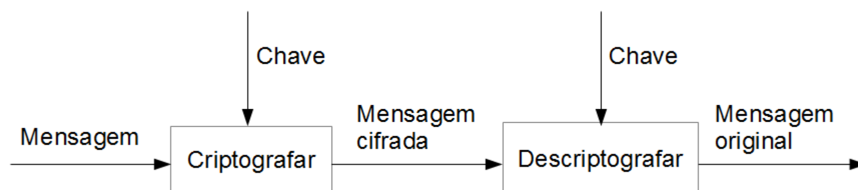

Figura 2.4: Processo de criptografar e descriptografar utilizando chave (adaptado de SCHNEIER (1996, p 3))

A chave usada para criptografar uma mensagem não necessariamente deve ser a mesma chave para descriptografar. O algoritmo de criptografar pode usar uma chave K_1, enquanto o algoritmo de descriptografar pode usar outra chave K_2. Portanto, temos:

$$E_{k_1}(M) = C$$

$$D_{k_2}(C) = M$$

$$D_{k_2}(E_{k_1}(M)) = M$$

Os dois tipos de algoritmos mais comuns são (SCHNEIER, 1996):

- DES[32]: é o algoritmo simétrico mais popular para computadores;
- RSA[33]: é o algoritmo de chave pública mais popular.

A seguir são descritos o algoritmo simétrico e o algoritmo de chave pública utilizados para criptografar e descriptografar uma mensagem.

[31]Tradução do termo em inglês *key*
[32]Iniciais de *Data Encryption Standard*
[33]Iniciais do nome dos criadores – Rivest, Shamir e Adleman

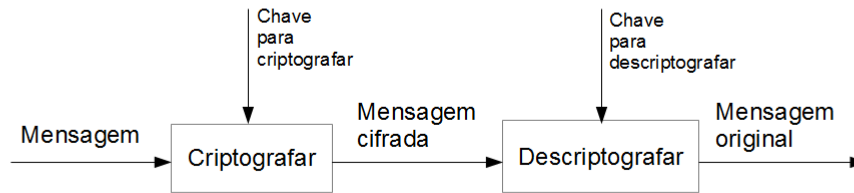

Figura 2.5: Processo de criptografar e descriptografar utilizando duas chaves diferentes (adaptado de SCHNEIER (1996, p 4))

2.3.2 Algoritmo simétrico

Algoritmo simétrico[34] é o algoritmo que usa uma mesma chave K para criptografar e descriptografar uma mensagem M. A segurança de um algoritmo simétrico está na chave utilizada para criptografar a mensagem. Divulgar a chave significa que qualquer pessoa pode descriptografar a mensagem. Para a comunicação permanecer em segredo, a chave deve permanecer secreta (SCHNEIER, 1996, p 17) (DELFS e KNEBL, 2002, p 11). O termo criptografia simétrica[35] significa que um algoritmo simétrico é usado para criptografar e descriptografar uma mensagem.

Criptografar e descriptografar uma mensagem via algoritmo simétrico é um processo que pode ser definido como:

$$E_k(M) = C$$

$$D_k(C) = M$$

Um problema desse tipo de algoritmo é como os participantes que se comunicam podem acordar sobre a chave K que será usada para criptografar e descriptografar a mensagem M. Tal problema estava sem solução até a descoberta do algoritmo de chave pública.

2.3.3 Algoritmo de chave pública

Um dos maiores marcos na criptografia moderna aconteceu quando Diffie e Helmann publicaram o trabalho *New Directions in Cryptography* (DIFFIE e HELLMAN, 1976). Este trabalho introduziu o conceito de algoritmo de chave pública. Algoritmo de chave pública, também chamado de algoritmo assimétrico[36] é o algoritmo que usa uma chave para criptografar e outra chave diferente para descriptografar uma mensagem. Qualquer pessoa pode usar uma chave para criptografar uma mensagem, mas só uma determinada pessoa pode descriptografar essa mensagem.

[34] Tradução do termo em inglês *symmetric algorithm*
[35] Tradução do termo em inglês *symmetric cryptography*
[36] Tradução do termo em inglês *public-key algorithm* ou *asymmetric algorithm*

A chave utilizada para criptografar a mensagem é chamada de chave pública[37], e a chave utilizada para descriptografar a mensagem é chamada de chave privada[38]. Deduzir a chave privada a partir da chave pública é matematicamente difícil (DIFFIE e HELLMAN, 1976). O termo criptografia de chave pública[39] significa que um algoritmo de chave pública é usado para criptografar e descriptografar uma mensagem. De todos os algoritmos de chave pública propostos, o algoritmo RSA (RIVEST *et al.*, 1978b) é o mais popular, fácil de entender e também de implementar.

O Algoritmo de chave pública não substitui o algoritmo simétrico, pelo contrário, ambos são complementares na configuração de uma comunicação segura. Algoritmos de chave pública são geralmente utilizados para definir as chaves que serão usadas num algoritmo simétrico. Para SCHNEIER (1996, p 38), há dois motivos para isso:

- Algoritmos de chave pública são lentos. Algoritmos simétricos são geralmente 1000 vezes mais rápidos;

- Algoritmos de chave pública são mais vulneráveis a ataques. Supondo que uma mensagem M pertença a um conjunto de n possibilidades, basta criptografar essas n possibilidades com a chave pública e comparar com a mensagem original criptografada. Esse ataque não determina a chave privada, mas descobre a mensagem M original.

Em conjunto, o algoritmo simétrico e o algoritmo de chave pública podem ser usados da seguinte maneira (SCHNEIER, 1996, p 38):

- Bob envia para Alice sua chave pública;

- Alice gera uma chave aleatória K e usa a chave pública de Bob para cifrá-la e enviá-la para Bob;

- Com sua chave privada, Bob descriptografa a mensagem enviada por Alice para recuperar a chave K;

- Ambos criptografam suas mensagens usando a mesma chave K.

Uma característica importante desse sistema é o fato de que a chave K é criada sob demanda, ou seja, uma nova chave é criada sempre que for necessário uma comunicação segura, e destruída quando a comunicação termina. Como a chave K não é guardada para uma comunicação futura, as chances de roubo dessa chave diminuem, já que a mesma permanece válida somente durante o período em que a comunicação está sendo praticada. Interessante observar que os passos realizados

[37]Tradução do termo em inglês *public key*

[38]Tradução do termo em inglês *private key*

[39]Tradução do termo em inglês *public-key cryptography*

por Alice e Bob constituem o que é chamado de protocolo criptográfico. Protocolo criptográfico será abordado em instantes. Antes, os algoritmos criptográficos de chave pública RSA e ElGamal serão descritos.

2.3.4 RSA

O algoritmo de chave pública RSA(RIVEST *et al.*, 1978b) é de fato o mais popular e também o mais fácil de entender e implementar. A segurança do RSA está na dificuldade de fatorar um número grande primo. Recuperar a mensagem original a partir da chave pública é equivalente a fatorar o produto de dois números primos, o que é um problema muito difícil, já que não são conhecidos algoritmos rápidos de fatoração.

Para gerar a chave pública, primeiro escolhe-se dois números primos p e q e calcula-se $n = pq$. Em seguida, calcula-se um inteiro positivo e que seja inversível módulo $\varphi(n)$, ou seja, e e $\varphi(n)$ são co-primos com $\text{mdc}(e, \varphi(n)) = 1$. A chave pública é composta pelo par (n, e).

Para criptografar a mensagem m, deve-se quebrar a mensagem m em blocos b que são menores que o valor numérico n. Para criptogragar o bloco b, calcula-se $c = b^e modn$. A mensagem m criptografada será o conjunto de blocos b criptografados.

Para descriptografar, calcula-se o número d que é o inverso de e em $\varphi(n)$. Em outras palavras, $d = e^{-1} mod \varphi(n)$ ou $d = e^{-1} mod((p-1)(q-1))$ A chave privada é composta pelo par (n, d). Para descriptografar c e recuperar o bloco b original, calcula-se $b = c^d modn$. Recuperando cada bloco b recupera-se a mensagem m original.

2.3.5 ElGamal

ElGamal é um algoritmo assimétrico descrito por ELGAMAL (1985) cuja segurança está baseada na dificuldade de resolver um logaritmo discreto. Lembrando que resolver um logaritmo discreto pode ser tão difícil quanto fatorar um número primo grande p como no algoritmo assimétrico RSA.

Para gerar as chaves do algoritmo criptográfico, primeiro define-se que G é um grupo cíclico de ordem q com elemento gerador g. Escolhe-se um número inteiro aleatório x que está entre $\{1, ..., q-1\}$. Calcula-se $h = g^x$. Além de tornar pública a estrutura do grupo G, a chave pública é composta por h, q, g, e a chave privada é o elemento x.

Para criptografar uma mensagem m, utiliza-se a chave pública (G,q,g,h). Escolhe-se um número inteiro aleatório r que está entre $\{1, ..., q-1\}$ e que seja coprimo com o número primo p, então calcula-se $a = g^r$ e $b = mh^r$. A notação para o texto criptografado é $(a, b) = (g^r, mh^r)$.

Para descriptografar (a, b), calcula-se $\frac{b}{a^x} = m$, ou $b * (a^{-1})^x = m$. O fator a^{-1} é o elemento inverso de a no grupo G.

Devido ao fator r aleatório no processo de criptografar, o texto criptografado para uma mensagem m não se repete, ou seja, se a mesma mensagem é criptografada mais de uma vez, não será obtido o mesmo bloco criptografado $(a,\ b)$.

O algoritmo ElGamal possui a propriedade de ser homomórfico multiplicativo e homomórfico aditivo. Criptografia homomórfica significa que é possível criptografar duas mensagens separadas, realizar uma operação com as duas mensagens criptografadas, descriptografar o resultado dessa operação e obter o mesmo resultado caso as duas mensagens originais fossem utilizadas. Por exemplo, ao criptografar o número 4, criptografar o número 7, realizar a operação de multiplicação nas duas mensagens criptografadas, descriptografar o resultado da multiplicação, o resultado obtido é o número 28, já que 4 * 7 = 28. O interessante é o fato de que quem descriptografa a mensagem obtém o resultado da operação, mas não tem conhecimento dos números originais que foram utilizados para gerar o resultado. A ideia de homomorfismo aplicado em criptografia foi primeiramente introduzida em (RIVEST *et al.*, 1978a) com o chamado homomorfismo privado, onde o autor argumenta que "parece que existe alguma função criptográfica que permite que o dado criptografado seja manipulado sem a necessidade de descriptografar o mesmo".

O homomorfismo multiplicativo do ElGamal pode ser verificado da seguinte forma:

$E(m_1) * E(m_2) = (g^{r1}, m_1 h^{r1})(g^{r2}, m_2 h^{r2}) = (g^{r1+r2}, (m_1 * m_2) h^{r1+r2}) = E(m_1 * m_2)$

Como resultado, as mensagens m_1 e m_2 são multiplicadas, pois as mensagens criptografadas foram multiplicadas. O homomorfismo aditivo pode ser obtido elevando o elemento gerador g pela mensagem m, resultando em g^m:

$E(g^{m_1}) * E(g^{m_2}) = (g^{r1}, g^{m_1} h^{r1})(g^{r2}, g^{m_2} h^{r2}) = (g^{r1+r2}, (g^{m_1+m_2}) h^{r1+r2}) = E(m_1 + m_2)$

No homomorfismo aditivo do ElGamal, a mensagem descriptografada estará na forma $g^{m_1+m_2}$. No entanto, isso significa que para achar $m_1 + m_2$ é necessário resolver o logaritmo discreto resultante de $g^{m_1+m_2}$, o que geralmente é uma tarefa difícil ou impraticável dependendo do tamanho em *bits* dos números do grupo G.

Além do algoritmo criptográfico ElGamal, outros algoritmos apresentam algum tipo de homomorfismo, com destaque para o RSA(RIVEST *et al.*, 1978b) que apresenta homomorfismo multiplicativo, Paillier (PAILLIER, 1999) com homomorfismo aditivo e multiplicativo, Goldwasser-Micali (GOLDWASSER e MICALI, 1982) com homomorfismo ou exclusivo (*XOR*) e Naccache-Stern (NACCACHE e STERN, 1997) com homomorfismo aditivo e multiplicativo. Como exemplo de aplicações do homomorfismo em criptografia, pode-se destacar o uso na computação em nuvem. Como

descrito em (TEBAA *et al.*, 2012), os dados são mantidos criptografados na nuvem garantindo a segurança da informação, sendo possível por meio da criptografia homomórfica executar operações com os dados criptografados. Como descrito no trabalho de CRAMER *et al.* (1997), o homomorfismo também tem aplicação em sistemas de votação eletrônica segura. Por exemplo, considerando uma votação onde a opção 1 (um) é a favor e a opção 0 (zero) é contra ao que está em pauta sendo votado, cada participante pode criptografar seu voto, de forma que por meio do homomorfismo aditivo seja possível somar os votos e obter o resultado final sem compromoter a segurança e a privacidade de quem votou.

2.3.6 O que é um protocolo criptográfico?

Protocolo é uma série de passos envolvendo dois ou mais participantes que tem por objetivo realizar uma tarefa. Uma "série de passos" significa que o protocolo tem início e fim. Envolvendo "dois ou mais participantes" significa que não existe protocolo com apenas um participante. E "tem por objetivo realizar uma tarefa" significa que o protocolo deve resolver algum problema (SCHNEIER, 1996) (MENEZES *et al.*, 1996).

Segundo SCHNEIER (1996), protocolos ainda apresentam outras características:

- Cada participante do protocolo deve conhecer o protocolo e todos os seus passos previamente;
- Cada participante deve concordar com o protocolo;
- O protocolo não deve ser ambíguo, ou seja, cada passo deve ser bem definido a fim de evitar interpretações erradas;
- O protocolo deve ser completo, ou seja, deve haver uma ação específica para cada situação possível.

Um protocolo criptográfico é um protocolo que usa criptografia, precisamente algum algoritmo criptográfico. Os participantes podem ser amigos que confiam um nos outros ou podem ser adversários. O objetivo de um protocolo criptográfico não é somente manter uma informação em segredo, mas sim evitar e detectar que alguém fora do protocolo escute o protocolo e ganhe informação ou que um dos participantes trapaceie. De outra forma, não deve ser possível fazer ou aprender mais do que especificado pelo protocolo (SCHNEIER, 1996).

Na descrição de um protocolo, é comum usar nomes reservados para os participantes do protocolo. Alice e Bob são os dois nomes mais comuns. Os nomes são usados por conveniência e também para personalizar a discussão, já que é mais fácil dizer que "Alice envia uma mensagem para Bob" do que "Participante A envia um

mensagem para o Participante B". Originalmente os nomes Alice e Bob apareceram no trabalho que descreve o algoritmo assimétrico RSA (RIVEST *et al.*, 1978b). Além desses principais nomes, há uma lista mais extensa que define outros nomes, onde cada nome desempenha um determinado papel no protocolo (SCHNEIER, 1996, p 23).

Os passos a seguir constituem um protocolo, onde Alice deseja enviar uma mensagem para Bob de forma segura. Nesse caso, o protocolo utiliza o algoritmo simétrico na comunicação (SCHNEIER, 1996, p 30):

1. Alice e Bob combinam sobre qual algoritmo simétrico será usado;
2. Alice e Bob combinam uma chave;
3. Alice criptografa sua mensagem utilizando o algoritmo e a chave que foram escolhidos;
4. Alice envia para Bob sua mensagem criptografada;
5. Bob descriptografa a mensagem utilizando o mesmo algoritmo e chave para recuperar a mensagem original.

O mesmo protocolo poderia ser feito utilizando o algoritmo de chave pública (SCHNEIER, 1996, p 32):

1. Alice e Bob definem as chaves públicas e privadas que serão utilizadas no protocolo;
2. Bob envia para Alice a sua chave pública;
3. Alice criptografa a mensagem utilizando a chave pública de Bob e envia a mensagem para Bob;
4. Bob descriptografa a mensagem usando sua chave privada.

Como descrito anteriormente, o algoritmo de chave pública não substitui o algoritmo de chave simétrica, mas é utilizado para gerar a chave que será utilizada no algoritmo simétrico.

2.3.7 *Bit Commitment*

Nessa subseção será descrito o protocolo *bit commitment* que é a base dos protocolos utilizados nessa dissertação. *Bit commitment* é um tipo de protocolo que permite a um participante se comprometer com uma mensagem (um ou mais *bits*) sem que a mesma seja revelada para os outros participantes. Além disso, a fim de evitar a manipulação ou alteração da mensagem original, o protocolo garante que

o participante que se comprometeu não consegue mudar sua mensagem antes que a mesma seja revelada (SCHNEIER, 1996, p 86).

O protocolo *bit commitment* é um protocolo de duas fases entre o emissor e o receptor. Na fase de comprometimento o emissor está comprometido com um valor específico que não poderá ser alterado. Além disso, o receptor não tem qualquer conhecimento sobre esse valor. Na fase de revelação, o emissor envia alguma informação extra para o receptor a fim de que o mesmo possa determinar o valor previamente comprometido (NAOR, 1991) (CRÉPEAU, 2012). O conceito de comprometimento com uma informação que estabelece uma comunicação justa e segura foi primeiramente formalizado por BRASSARD *et al.* (1988).

Como exemplo, considere que Alice e Bob estão jogando par ou ímpar, sendo que Alice escolheu par e Bob escolheu ímpar. Mentalmente, Alice escolhe um número. Da mesma forma, Bob também o faz. Como Bob confia em Alice, Bob fala primeiro o seu número e pergunta a Alice qual número ela pensou. Se Alice for justa, ela diz o seu verdadeiro número e ambos conseguem saber se a soma dos dois números é par ou ímpar. No entanto, caso Alice seja uma trapaceira, ela pode mudar o número que ela pensou originalmente para um número que, somado com o número de Bob, seja par. Nesse caso, Bob não saberia que foi trapaceado.

Como forma de contornar esse problema, o protocolo *bit commitment* permite que Bob se comprometa com um número sem que esse número seja revelado para Alice. Por exemplo, Bob escreve o seu número em um pedaço de papel e o sela dentro de um envelope. Bob envia o envelope para Alice, sendo que Alice não consegue ver o que está dentro do envelope. Em seguida, Alice revela o seu número para Bob. Agora, ambos podem abrir o envelope selado e verificar se a soma dos dois números é par ou ímpar.

Formas de garantir o *Bit Commitment*

A fim de garantir que um participante se comprometa com uma informação e que a mesma não possa ser alterada posteriormente, o uso da criptografia simétrica ou de função de mão única geralmente são usadas para implementar o protocolo de *bit commitment*. A seguir serão descritos os protocolos usando ambas abordagens.

Usando a criptografia simétrica, o protocolo *bit commitment* pode ser construído da seguinte forma (SCHNEIER, 1996):

1. Bob gera um ou mais *bits* aleatórios R e envia para Alice;

   R

2. Alice cria uma mensagem composta por seus *bits* de interesse mais a mensagem R enviada por Bob. Ela criptografa a mensagem com alguma chave aleatória K e envia a mensagem criptografada para Bob;

$E_K(R, b)$

3. Alice envia para Bob a chave K;

4. Bob descriptografa a mensagem para revelar a mensagem. Para validar a mensagem que Alice enviou, Bob verifica se sua mensagem R está presente na mensagem descriptografada.

Se o protocolo não usar a mensagem R de Bob, Alice poderia, depois do passo 2, testar mais de uma chave para descriptografar a sua mensagem. Cada chave resultaria em uma mensagem diferente. Portanto, Alice poderia trapacear Bob enviando-o uma chave que resultaria em uma mensagem diferente da que ela escolheu originalmente. Acrescentando a mensagem R, Alice teria que encontrar uma chave que, além de trocar a sua mensagem original para outra mensagem de seu interesse, também resultasse na mensagem R que Bob escolheu.

Antes de descrever o protocolo *bit commitment* via função de mão única, é necessário saber o que é uma função de mão única. Função de mão única[40] é uma função fácil de ser calculada mas extremamente difícil de ser revertida. Dado um valor qualquer x, é fácil calcular $f(x)$, mas dado $f(x)$ é difícil calcular x (GOLDREICH e LEVIN, 1989). Difícil de ser revertida significa que milhões de anos seriam necessários para reverter $f(x)$ mesmo se todos os computadores do mundo fossem utilizados. Matematicamente ainda não foi provado a existência de uma função de mão única, mas existem funções que são usadas como uma (SCHNEIER, 1996). Idealmente, em funções de mão única, $f(x) \mathrel{!=} f(x')$ se $x \mathrel{!=} x'$.

Usando uma função de mão única H, o protocolo *bit commitment* pode ser construído da seguinte forma:

1. Alice gera duas strings aleatórias, R_1 e R_2;

   R_1, R_2

2. Alice cria uma mensagem composta por R_1 e R_2 mais o *bit* b que ela deseja se comprometer;

   (R_1, R_2, b)

3. Alice computa o valor da função de mão única para a mensagem e a envia juntamente com uma das mensagens aleatórias para Bob;

   $H(R_1, R_2, b), R_1$

4. Alice envia para Bob a mensagem original;

   (R_1, R_2, b)

[40] Tradução do termo em inglês *one-way function*

5. Bob calcula o valor da função única da mensagem e compara com a mensagem original enviada por Alice. Bob valida essa mensagem comparando o valor de R_1 com o valor de R_1 recebido no passo 3.

Nesse protocolo Bob não precisa enviar nenhuma mensagem para Alice. Alice se compromete com seu *bit b* quando envia o resultado da função única juntamente com a mensagem R_1 no passo 3. Para Alice trapacear alterando o *bit b* para *b'*, ela teria que achar outra mensagem R_2' onde $H(R_1, R_2', b') = H(R_1, R_2, b)$, o que praticamente impede que Alice mude a sua mensagem depois de enviá-la para Bob.

2.3.8 *Secret Sharing*

Secret sharing, ou compartilhamento de segredo é um protocolo que tem o propósito de manter uma mensagem em segredo entre um grupo de participantes. Esse protocolo, inventado independentemente por SHAMIR (1979) e BLAKLEY (1979), quebra a mensagem em pedaços, onde cada pedaço sozinho não significa nada, mas quando os pedaços são reunidos a mensagem é reconstruída. Em geral um participante confiável gera um segredo x e distribui uma parte desse segredo para os n participantes. Se o segredo puder ser reconstruído a partir de k participantes, esse sistema é chamado de (k,n)-*threshold*. Por exemplo, se 10 participantes compartilham um segredo, e caso seja necessário apenas 4 participantes para revelar o segredo, esse sistema é chamado de (4,10)-*threshold*. No entanto, k - 1 participantes não podem reconstruir o segredo, ou seja, 3 ou menos participantes não são suficientes para reconstruir o segredo.

Para descrever o protocolo proposto por SHAMIR (1979), seja o segredo $D = D_1, ..., D_n$, onde n é o número de participantes. Seja também um número primo p maior que D e n. O protocolo (k,n)-*threshold* é um protocolo baseado em interpolação polinomial: dado k coordenadas (x_i, y_i),...,(x_k, y_k) com diferentes valores para cada x_i, há um polinômio $f(x)$ de grau k - 1 tal que $f(x_i) = y_i$ para todo i. Para dividir o segredo D em D_i partes, escolhe-se um polinômio aleatório $f(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$ de grau k - 1, onde $a_0 = D$ e calcula-se $D_1 = f(1)$, ... , $D_i = f$(i),..., $D_n = f$(n). Os coeficientes $a_1, ..., a_{k-1}$ são inteiros escolhidos tal que 0 $\leqslant$ a $< p$ e os valores $D_i, ..., D_n$ são calculados módulo p. Dessa forma, dado um subconjunto de k do conjunto de valores D_i é possível achar os coeficientes de *f(x)* por interpolação polinomial e então calcular $D = f(0)$.

Seguindo conceitualmente o protocolo acima, PEDERSEN (1991) propõe um protocolo distribuído para compartilhar um segredo. Pode-se destacar as seguintes características do protocolo:

- Não depende de uma parte confiável como no protocolo descrito acima, ou seja, a função polinomial $f(x)$ não é escolhida por um único participante de

confiança dos outros participantes, mas sim gerada a partir da colaboração de todos os participantes;

- O pedaço do segredo que um participante recebe pode ser validado, ou seja, pode-se garantir que o segredo recebido está correto. Essa propriedade é importante porque o segredo não é mais calculado por uma parte confiável;
- A segurança está baseada na dificuldade de calcular um logaritmo discreto.

Assumindo que p e q são número primos e que $q \mid p$ - 1, G_q é um grupo de ordem q e g um gerador de G_q, o protocolo pode ser descrito da seguinte forma:

1. Cada participante P_i escolhe um polinômio aleatório $f_i(z)$ em G_q de grau t:

   $f_i(z) = a_{i0} + a_{i1}z + ... + a_{it}z^t$

   P_i envia para cada participante $X_{ik} = g^{a_{ik}}$ mod p para k = 0,...,t. Seja o valor a_{i0} representado por x_i e X_{i0} por h_i. Cada P_i calcula o segredo parcial $x_{ij} = f_i(j)$ mod q para j = 1,...,n e envia x_{ij} de forma segura para o jogador P_j.

2. Cada jogador P_j verifica os segredos parciais que recebeu dos outros jogadores ao checar, para i = 1,...,n:

   $g^{x_{ij}} = \prod_{k=0}^{t}(X_{ik})^{j^k} \bmod p$

   Se algum teste falhar para um índice i, o jogador P_j envia para todos os outros jogadores uma denúncia contra o jogador P_i.

3. Se mais do que t jogadores denunciarem o jogador P_i, então o jogador P_i é banido.

4. O valor público h é calculado como $h = \prod h_i \bmod p$. O valor secreto compartilhado x não é calculado por nenhum jogador, sendo $x = \sum x_i \bmod q$.

Esse protocolo tem um papel fundamental na geração e distribuição das chaves (GDC) do algoritmo ElGamal utilizado no protocolo *Mental Poker* descrito na Seção 2.3.10.

2.3.9 *Fair Coin Flipping*

Alice e Bob querem jogar cara ou coroa sem uma moeda física. Alice propõe a Bob que os dois pensem em um *bit* aleatório. O resultado será o ou exclusivo (*XOR* – *exclusive-or*) entre esses dois *bits*. Alice e Bob podem determinar que o *bit* 0 é cara e que o *bit* 1 é coroa. O protocolo que descreve esse cenário pode ser dessa forma:

1. Alice pergunta a Bob qual o *bit* que ele pensou;

2. Bob fala para Alice o *bit* que ele pensou;

3. Alice pega a resposta de Bob e calcula o *XOR* de seu *bit* com o *bit* enviado por Bob;

4. Alice revela o resultado para Bob.

Pode-se perceber que esse cenário descrito por esse protocolo permite que Alice trapaceie. Caso Alice e Bob sejam justos e verdadeiros, o protocolo funciona. Caso Alice e Bob não se conheçam ou sejam adversários, Alice pode trapacear dizendo um *bit* cujo resultado final seja a seu favor.

Endereçando o problema acima, BLUM (1982) introduziu o problema onde duas pessoas querem jogar moeda por telefone, além de propor uma solução usando um protocolo de *bit commitment*. O protocolo segue a seguinte estrutura:

1. Alice compromete seu *bit* aleatório com Bob utilizando um protocolo de *bit commitment*;

2. Bob revela seu *bit* para Alice;

3. Alice revela o *bit* de resultado para Bob. Bob vence caso tenha acertado o *bit* de resultado.

No passo 2, Alice já conhece o resultado final, porém não pode mudá-lo, pois no passo 1 comprometeu-se com Bob enviando seu *bit* aleatório. No passo 3, Bob verifica se o valor comprometido por Alice no passo 1 é igual ao valor gerado por um protocolo de *bit commitment* sobre o valor revelado por Alice no passo 3.

Um protocolo que se propõe a resolver o problema de jogar moeda sem uma moeda física deve apresentar as seguinte propriedades (BLUM, 1982):

1. Se qualquer um dos participantes do protocolo não pegar o outro trapaceando, então ele tem certeza de que a probabilidade de sair cara ou coroa é de 50%;

2. Se qualquer um dos participantes do protocolo pegar o outro trapaceando, então deve ser possível provar que o outro trapaceou;

3. Depois que Bob jogou sua moeda, Alice sabe o resultado, mas Bob ainda não tem ideia do resultado;

4. Depois que ambos jogaram a moeda, Alice deve ser capaz de provar para Bob qual o resultado da moeda.

Uma forma de implementar um protocolo que possua as propriedades descritas anteriormente é usar uma função de mão única (SCHNEIER, 1996, p 83):

1. Alice escolhe um número aleatório x. Assumindo que $f(x)$ é uma função de uma mão única, Alice calcula $y = f(x)$;
2. Alice envia y para Bob;
3. Bob adivinha se x é par ou ímpar e envia sua escolha para Alice;
4. Se Bob acertar na sua escolha, o resultado da moeda será cara. Se a escolha for incorreta, o resultado da moeda será coroa. Alice anuncia o resultado e envia o número x para Bob;
5. Bob confirma que $y = f(x)$.

A primeira fase do comprometimento acontece no passo 2, quando Alice envia o resultado da função de mão única para Bob. Nesse ponto, Alice não pode mais mudar de ideia sobre o valor original de x, pois Bob irá conferir no passo 5, por meio da aplicação da função de mão única com o valor x enviado por Alice, se o resultado é igual ao valor de y enviado por Alice no passo 2.

O mesmo protocolo pode ser construído usando a criptografia de chave pública (SCHNEIER, 1996, p 84):

1. Alice e Bob geram suas chaves pública e privada;
2. Alice gera duas mensagens, uma indicando cara e outra indicando coroa. Essas mensagens devem conter alguma sequência aleatória de *bits* que será usada para verificar a autenticidade do protocolo. Alice criptografa ambas as mensagens com sua chave pública e as envia para Bob;

   $E_A(M_1), E_A(M_2)$
3. Como Bob não pode ler o conteúdo da mensagem, ele escolhe uma aleatoriamente. Ele criptografa a mensagem escolhida com sua chave pública e envia o resultado para Alice;

   $E_B(E_A(M))$, onde M é M_1 ou M_2
4. Alice, que não pode ler a mensagem recebida, descriptografa a mensagem com sua chave privada e envia o resultado para Bob;

   $D_A(E_B(E_A(M))) = E_B(M_1)$ se $M = M_1$ ou $E_B(M_2)$ se $M = M_2$
5. Bob descriptografa a mensagem com sua chave privada para revelar o resultado final. Bob envia a mensagem descriptografada para Alice.

   $D_B(E_B(M_1)) = M_1$ ou $D_B(E_B(M_2)) = M_2$

6. Alice recebe o resultado e verifica se a mensagem aleatória criada no passo 2 está correta;

7. Alice e Bob revelam as suas chaves para ambos verificarem se ninguém trapaceou.

Alice poderia gerar duas mensagens iguais (que poderiam ser duas caras ou duas coroas), mas no passo 7 Bob descobriria essa trapaça. Alice poderia usar outra chave para descriptografar a mensagem no passo 4, mas Bob iria perceber que a mensagem gerada no passo 5 está sem sentido. Por outro lado, Bob poderia forçar coroa criptografando incorretamente no passo 3, mas Alice iria perceber verificando a mensagem aleatória no passo 6. Bob poderia alegar que não consegue descriptografar a mensagem no passo 5 por causa de uma eventual trapaça de Alice, mas essa tentativa de fraude seria descoberta no passo 7. Dessa forma, o protocolo mostra-se seguro contra trapaças que podem acontecer de ambas as partes.

Pode-se imaginar que uma moeda é um dado com duas faces. Seguindo os mesmos princípios do protocolo *fair coin flipping*, pode-se estender esse protocolo para criar um protocolo de jogar dado de uma forma segura entre dois participantes que não confiam um no outro. Se não houver um protocolo, um jogador pode alegar que tirou 6 e 6, e o outro jogador não irá acreditar ou não terá como provar a trapaça.

O protocolo descrito a seguir integra a biblioteca JGAMMON (2012), biblioteca que implementa o jogo gamão e que está escrita na linguagem Java[41]:

1. Cada participante gera um número aleatório de 64 *bits*: R_1 e R_2;

2. Cada um calcula uma função de mão única para o número aleatório. Cada um envia o resultado da função de mão única para o outro. Nesse passo, utiliza-se a função SHA[42] como função de mão única. SHA(R_1) e SHA(R_2);

3. Cada participante envia o número aleatório gerado no passo 1;

4. Cada participante verifica se o cálculo da função de mão única de R_i enviado no passo anterior é igual ao resultado da função de mão única enviada no passo 2;

5. Ambos os participantes podem gerar o resultado do dado da seguinte forma: (primeiro_byte(R_1) + primeiro_byte(R_2)) *mod* 36.

[41]http://java.com/en/download/index.jsp

[42]Iniciais de *Secure Hash Algorithm*, SHA foi projetado pelo NIST - *National Institute of Standards and Technology* - em parceria com a NSA - *National Security Agency* - como padrão no uso de assinatura digital.

Esse protocolo tem as mesmas propriedades do protocolo *fair coin flipping* que utiliza a função de mão única. Ambos os participantes não podem trapacear, pois o número que cada um escolheu para gerar o resultado final foi comprometido no passo 2. Outro protocolo semelhante utilizando a linguagem de programação E[43] também segue o mesmo princípio do protocolo descrito anteriormente (COMMUNITIES, 2012).

2.3.10 *Mental Poker*

Num jogo de cartas, jogadores estão sentados ao redor de uma mesa jogando pôquer. Nesse cenário é relativamente factível ver se alguém está trapaceando. No entanto, se os jogadores não estão fisicamente presentes e resolvem jogar pôquer via e-mail ou via telefone, ou seja, trocando mensagens em vez de cartas, como garantir que essas mensagens que representam as cartas do baralho são distribuídas da mesma forma para todos os jogadores? Como garantir que nenhum jogador trapaceie observando as cartas do outro jogador?

O termo *Mental Poker* significa jogar cartas sem um baralho via um canal de comunicação, como por exemplo a internet, sem a intervenção de uma terceira parte em que todos confiam e que tem a função de facilitar a interação entre os jogadores. Segundo FORTUNE e MERRITT (1984) a primeira discussão sobre *Mental Poker* foi em 1933 quando Niels Bohr tentou jogar com seus amigos, porém sem sucesso. A primeira tentativa mais séria e fundamentada ocorreu em 1979, quando SHAMIR *et al.* (1979) chegaram a um resultado paradoxal. Segundo os autores, o jogo é justo se estiver de acordo com as seguintes restrições:

1. Cada jogador deve conhecer as cartas em sua mão, mas não deve ter informações das cartas dos outros jogadores;

2. O método de distribuir as cartas deve assegurar que as mãos são disjuntas, ou seja, uma mesma carta não pode estar em mais de uma mão;

3. A mão de cada jogador é formada com a mesma probabilidade;

4. Durante o jogo, os jogadores podem pegar as cartas restantes do baralho, ou revelar alguma carta sem comprometer a segurança das cartas da sua mão.

De fato SHAMIR *et al.* (1979) provaram matematicamente que é impossível conceber um *Mental Poker* com os requisitos acima, mas forneceram uma solução prática para o problema com um protocolo criptográfico. A seguir o protocolo para dois jogadores é descrito (SCHNEIER, 1996, p 92):

[43]http://erights.org/

1. Alice e Bob, cada um gera o seu par de chaves pública e privada;

2. Alice gera 52 mensagens, uma para cada carta do baralho. Cada mensagem deve conter uma *string* aleatória que servirá para verificar a autenticidade das mensagens. Alice criptografa todas as mensagens com sua chave pública e as envia para Bob;

   $E_A(M_n)$

3. Bob, que não pode ler as mensagens, escolhe cinco mensagens ao acaso. Ele criptografa essas cinco mensagens com sua chave pública e as envia de volta para Alice;

   $E_B(E_A(M_n))$

4. Alice, que não pode ler as mensagens, descriptografa com sua chave privada e as envia de volta para Bob;

   $D_A(E_B(E_A(M_n))) = E_B(M_n)$

5. Bob descriptografa suas mensagens com sua chave privada para revelar suas cartas;

   $D_B(E_B(M_n)) = M_n$

6. Bob escolhe mais cinco cartas ao acaso das 47 cartas restantes e as envia para Alice;

   $E_A(M_n)$

7. Alice descriptografa as mensagens com sua chave privada para revelar suas cartas;

   $D_A(E_A(M_n)) = M_n$

8. No final, Alice e Bob revelam suas cartas e suas chaves para que todos tenham certeza de que ninguém trapaceou.

Esse protocolo pode ser estendido para três ou mais jogadores. Tanto esse protocolo como o descrito a seguir usam criptografia comutativa[44], isto é, se alguma mensagem é criptografada mais de uma vez, a ordem em que essa mensagem é descriptografada não importa. A seguir é descrito o protocolo estendido para três jogadores (SCHNEIER, 1996, p 93):

1. Alice, Bob e Carol, cada um gera seu par de chaves pública e privada;

[44] Tradução do termo em inglês *cryptographic commutative*

2. Alice gera 52 mensagens, uma para cada carta do baralho. Cada mensagem deve conter uma string aleatória que servirá para verificar a autenticidade das mensagens. Alice criptografa todas as mensagens com sua chave pública e as envia para Bob;

   $E_A(M_n)$

3. Bob, que não pode ler as mensagens, escolhe cinco mensagens ao acaso. Ele criptografa essas cinco mensagens com sua chave pública e as envia de volta para Alice;

   $E_B(E_A(M_n))$

4. Bob envias as outras 47 mensagens para Carol;

   $E_A(M_n)$

5. Carol, que não pode ler as mensagens, escolhe cinco mensagens ao acaso. Ela criptografa essas cinco mensagens com sua chave pública e as envia para Alice;

   $E_C(E_A(M_n))$

6. Alice, que não pode ler as mensagens, descriptografa com sua chave privada e as envia de volta para Bob e Carol;

   $D_A(E_B(E_A(M_n))) = E_B(M_n)$

   $D_A(E_C(E_A(M_n))) = E_C(M_n)$

7. Bob e Carol descriptografam suas mensagens com sua respectiva chave privada para revelar suas cartas;

   $D_B(E_B(M_n)) = M_n$

   $D_C(E_C(M_n)) = M_n$

8. Carol escolhe mais cinco cartas ao acaso das 42 cartas restantes e as envia para Alice;

   $E_A(M_n)$

9. Alice descriptografa as mensagens com sua chave privada para revelar suas cartas;

   $D_A(E_A(M_n)) = M_n$

10. No final, Alice, Bob e Carol revelam suas cartas e suas chaves para que todos tenham certeza de que ninguém trapaceou.

O protocolo proposto acima, no entanto, não oferece confidencialidade, requerendo que cada jogador revele a sua mão ao fim do jogo (passo 10) para assegurar que ninguém trapaceou. Isso significa que a estratégia do jogador passa a ser conhecida pelos seus oponentes, o que geralmente não é desejável acontecer na maioria de jogos de cartas que envolvem dinheiro. Por exemplo, um jogador que blefou deve mostrar suas cartas e revelar sua estratégia para os outros jogadores da mesa. Idealmente, o passo 10 não seria necessário. Como colocado por SCHNEIER (1996, p 93), somente é interessante saber se o ganhador trapaceou. Todos os outros jogadores podem trapacear desde que percam o jogo.

Além disso, LIPTON (1981) provou que uma pequena quantidade de informação é vazada nesse algoritmo que utiliza a criptografia RSA. Se a representação binária da carta é um resíduo quadrático, então a carta criptografada também será um resíduo quadrático. Essa propriedade pode ser usada, por exemplo, para marcar todas as cartas de um determinado naipe.

O estudo de protocolos que endereçam o problema do *Mental Poker* evoluiu com os trabalhos apresentados por Claude Crépeau. Para CRÉPEAU (1985), um jogo de pôquer deve alcançar as seguintes condições:

1. Unicidade das cartas;
2. Distribuição uniforme e aleatória das cartas;
3. Ausência de uma terceira parte confiável;
4. Detectar com alta probabilidade qualquer trapaça;
5. Confidencialidade completa das cartas;
6. Minimizar os efeitos de aliança entre jogadores fora do protocolo;
7. Confidencialidade completa da estratégia.

Em seu trabalho, CRÉPEAU (1985) propõe um protocolo que satisfaz os seis primeiros requisitos, mas não alcança o sétimo requisito - confidencialidade completa da estratégia. Para iniciar o protocolo, suponha que $P_1, P_2, ..., P_N$ desejam jogar pôquer e que $DECK = \{1, 2,..., 52\}$. Assuma também a correspondência entre o baralho padrão e o conjunto $\{1, 2,..., 52\}$. Cada P_i escolhe uma permutação π_i de $\{1, 2,..., 52\}$ e mantém em segredo. O baralho embaralhado será a composição de funções das permutações, ou seja, $\pi_N...\pi_2\pi_1$.

Para um jogador P_i escolher uma carta, o mesmo escolhe um valor k em $DECK$ que ninguém tenha escolhido, e gera a sua carta calculando $\pi_N...\pi_2\pi_1(k)$. Como as permutações são mantidas em segredo, o valor de uma permutação é obtido usando o protocolo Escondendo-Revelando[45] proposto no mesmo trabalho. Esse protocolo

[45]Tradução para o termo em inglês *Hiding-Revealing protocol*

permite que uma parte A escolha um valor de um conjunto conhecido pela parte B sem que a parte B saiba qual valor a parte A escolheu. Isso permite P_i obter os valores $\pi_1(k), \pi_2\pi_1(k)$ até $\pi_N...\pi_2\pi_1(k)$ dos outros jogadores. O jogo prossegue com os jogadores obtendo cartas dessa forma. Contudo, algum jogador pode trapacear ao computar $\pi_N...\pi_2\pi_1(k^{'})$ para algum $k^{'} \in DECK$ que não o pertence. Com isso esse jogador pode ver cartas que estão na mão de outro jogador ou que estão no baralho. Para verificar esse tipo de trapaça, o protocolo exige que cada jogador revele sua permutação π_i no final do jogo, o que revela a mão de cada jogador. Em outro trabalho que tem por base o protocolo apresentado, CRÉPEAU (1986) consegue satisfazer o sétimo requisito, tornando-se a primeira solução completa para o problema do *Mental Poker*. Nesse novo protocolo, a ideia é adicionar informações distintas e aleatórias para cada valor secreto em $\pi_1, \pi_2, ..., \pi_N$, onde a informação adicionada é grande o suficiente para não ser adivinhada. Quando um jogador ler o valor da permutação do outro jogador, também será lida essa informação. Essas informações serão posteriormente reveladas pelos jogadores, e se todas forem diferentes, ninguém trapaceou. Sendo s um parâmetro seguro escolhido pelos jogadores, P_i escolhe $\tau_{i,j}$ com $1 \leq j \leq$ i - 1, e alguns *arrays* de tamanho 52 com strings aleatórias de tamanho s. Para $k \in DECK$, $\tau_{i,j}(k)$ é chamado de registro de $\pi_i(k)$ para P_j. Sempre que P_j ler $\pi_i(k)$, ele receberá o valor $\tau_{i,j}(k)$ correspondente e não um outro valor $\tau_{i,j}(k^{'})$. Se P_j tentar trapacear lendo algum $\pi_i(k^{'})$ em vez do valor correto $\pi_i(k)$, ele também receberá o valor $\tau_{i,j}(k^{'})$ em vez de $\tau_{i,j}(k)$. Dessa forma, se um jogador desonesto ler uma carta que pertence a outro jogador P_i, ele receberá o mesmo registro, o que acusa a trapaça ao fim do jogo. Embora o protocolo funcione em teoria, EDWARDS (1994) provou não funcionar na prática. Três computadores com arquitetura SPARC demoraram oito horas para embaralhar um baralho.

CRÉPEAU (1986) define protocolos para preparar o jogo, pegar uma carta, detectar eventuais trapaças, restaurar o registro das jogadas, descartar e revelar uma carta. SCHINDELHAUER (1998) argumenta que esses protocolos apresentam a limitação de que devem seguir uma determinada sequência. Por exemplo, o baralho deve ser embaralhado somente no início do jogo. Além disso, SCHINDELHAUER (1998) aponta outras limitações: as cartas devem ser únicas no baralho, não pode reutilizar cartas que foram descartadas e não pode inserir uma nova carta no baralho. Em suma, os protocolos definidos por CRÉPEAU (1986) são suficientes para jogos de pôquer, mas não satisfazem operações mais genéricas em outros tipos de jogos.

Dessa forma, SCHINDELHAUER (1998) modifica e estende muitas ideias e técnicas de CRÉPEAU (1986) e propõe um conjunto de operações genéricas que fazem sentido na maioria dos jogos. As operações são aplicadas em uma única carta, no baralho ou num conjunto de cartas. As operações aplicadas nas cartas são escolher uma carta do baralho mostrando ou não a carta para os outros jogadores e

também revelar uma carta da mão do jogador para os outros jogadores. As operações aplicadas no baralho compreendem a criação do baralho, embaralhar, inserir carta no baralho e particionar o baralho, ou seja, escolher k cartas do topo e colocar no fim do baralho. A ideia é compartilhar a carta entre todos os jogadores, ou seja, cada carta é criptografada com a chave pública de cada jogador, sendo necessário a participação de todos os jogadores para gerar ou revelar uma carta. O tamanho de uma carta cresce linearmente com o número de jogadores. A cada operação é realizada uma verificação de correção, garantindo que ninguém trapaceou. Em vez de verificar a correção das operações ao final do jogo, a verificação ocorre a cada operação. BARNETT e SMART (2003), baseado no trabalho e nas operações de cartas descritas por SCHINDELHAUER (1998), propõe um *framework* abstrato composto por quatro protocolos, com um protocolo de geração de chaves, protocolo de codificação, protocolo de codificação de uma carta que já está codificada e protocolo de decodificação. O *framework* proposto está baseado no logaritmo discreto e foi implementado com o ElGamal(ELGAMAL, 1985). A principal vantagem apresentada por BARNETT e SMART (2003) é a redução do tamanho da carta codificada que, ao contrário da proposta de SCHINDELHAUER (1998), o tamanho da carta em *bits* é independente do número de jogadores. Entre as operações possíveis da abordagem de BARNETT e SMART (2003) está a criação de uma carta aberta, criação de uma carta fechada, criação de uma carta aleatória a partir do baralho, publicação de uma carta fechada para os outros jogadores, criação de um baralho, embaralhar um baralho e partir o baralho ao meio, ou seja, colocar x cartas da base no topo do baralho.

Os trabalhos apresentados até agora sobre *Mental Poker* de fato são modelos abstratos e não foram implementados na prática, ou quando foram implementados não foram práticos como mostrou o trabalho de EDWARDS (1994) que demorou oito horas para embaralhar um baralho. No entanto, o trabalho apresentado por STAMER (2005) parece ser de fato a primeira implementação prática que não requer uma terceira parte confiável e que mantém a confidencialidade da estratégia dos jogadores. Nesse trabalho o autor incorporou o sistema de criptografia proposto por BARNETT e SMART (2003), permitindo que o tamanho em *bits* da carta não dependa do número de jogadores. Com isso, o autor criou uma implementação eficiente do *framework* proposto por SCHINDELHAUER (1998).

STAMER (2005) desenvolveu uma biblioteca chamada LibTMCG[46] em C++ que tem como objetivo a criação de jogos de cartas *peer-to-peer* de uma forma segura. A biblioteca tem aproximadamente 7300 linhas de código e está sob a licença GPL[47].

[46] O endereço que contém o projeto LibTMCG é http://www.nongnu.org/libtmcg/.

[47] A Licença Pública Geral, do inglês *GPL General Public License* é uma licença de *software* livre que permite ao usuário executar, copiar, distribuir, estudar e alterar o *software*. Detalhes da licença GNU pode ser encontrado em http://www.gnu.org/licenses/gpl.html.

Como exemplo de uso da biblioteca, o autor cita o desenvolvimento de uma rede *peer-to-peer* chamada *SecureSkat*(STAMER, 2013) que implementa o jogo de cartas Skat[48].

Apesar de realizar uma implementação prática dos protocolos criptográficos abstratos de *Mental Poker*, STAMER (2005) ressalta que em jogos com muitos jogadores ou com um baralho grande, por exemplo um baralho de pôquer com 52 cartas, as técnicas e operações utilizadas na biblioteca LibTMCG ainda são muito custosas. A seguir é descrito um protocolo mais novo e com uma abordagem diferenciada em relação aos protocolos descritos nessa seção. De fato o protocolo descrito a seguir será o protocolo implementado e estendido no módulo que gerencia cartas da API proposta nesse trabalho.

Abordagem orientada ao pôquer

Com base nos protocolos descritos anteriormente, GOLLE (2005) propõe um novo protocolo para embaralhar e distribuir cartas. O protocolo proposto utiliza duas características de jogos de cartas, em especial jogos de pôquer, que são negligenciadas por protocolos genéricos de cartas:

1. cartas em jogos de pôquer são distribuídas em rodadas, com apostas entre as rodadas em vez de serem distribuídas de uma só vez;
2. o número total de cartas utilizadas num jogo de pôquer é pequeno e depende do número de jogadores. O número total de cartas utilizadas num jogo geralmente não é maior que a metade do baralho.

A ideia do protocolo é distribuir o custo de embaralhar e distribuir as cartas ao longo das rodadas em vez de realizar essa tarefa num único momento com tempo e custo de processamento elevado. Comparado com outros protocolos que embaralham todo o baralho de uma só vez, GOLLE (2005) argumenta que seu protocolo diminui a latência[49] e o custo computacional por realizar essa tarefas de forma fragmentada ao longo do jogo. Segundo o autor, a partir de uma estimativa calculada analiticamente por meio do número de modulação exponencial, o custo computacional para embaralhar e distribuir as cartas num jogo de Texas Hold'em[50] entre 5 jogadores foi 66% menor comparado com outros protocolos mais genéricos. Também foi levantado que a latência para a configuração inicial do jogo - etapa que distribui as cartas para compor a mão inicial de cada jogador - foi 85% menor.

[48]Skat é um jogo de cartas popular da Alemanha jogado com 3 jogadores e um baralho com 32 cartas.

[49]O termo latência no contexto do protocolo é entendido como o tempo de comunicação entre os jogadores.

[50]Texas Hold'em é estilo mais popular de jogo de pôquer geralmente jogado com dois a cinco jogadores.

Conceitualmente, as cartas no protocolo são representadas pelo intervalo [1,...,52]. Uma carta c aleatória é gerada juntamente pelos jogadores ao calcular a criptografia *E(c)* de $c \in$ a [1,...,52]. Os jogadores comparam *E(c)* com cada carta criptografada *E(c1),...,E(ct)* que foi distribuída durante o jogo. Se *E(c)* for igual a uma carta criptografada, *E(c)* é descartada e uma nova carta é gerada. Caso a carta ainda não tenha saído no jogo e essa carta é para o jogador P, os outros jogadores ajudam o jogador P a descriptografar *E(c)* de forma que a carta c seja conhecida apenas por P. Percebe-se que a ação de embaralhar e distribuir as cartas ocorrem em paralelo a cada carta que é gerada.

Em contrapartida, o protocolo proposto não é adequado para distribuir todas as cartas do baralho, pois a medida que aumenta o número de cartas que são geradas, o número de colisões também aumenta. O número de colisões esperado para distribuir a cartas é de aproximadamente $\frac{1}{52}(\frac{a(a-1)}{2})$ colisões para $a \leq 52$.

Assumindo que k seja o número de jogadores e que num cenário estressante k - 1 jogadores estão trapaceando, o protocolo proposto assegura as seguintes propriedades:

- Correção[51]: A cada carta distribuída, seja para quem está trapaceando ou não, a carta é gerada de forma aleatória a partir do conjunto de cartas;
- Privacidade[52]: O jogador(es) que está(ão) trapaceando não obtém(êm) nenhuma informação adicional sobre as cartas dos outros jogadores que não estão trapaceando;
- Robustez[53]: O protocolo permite que um jogador desista do jogo, mas previne que um jogador impeça outro jogador de continuar jogando.

Com um baralho de 52 cartas como exemplo, a seguir será descrito o protocolo proposto por GOLLE (2005) que utiliza o sistema de criptografia ElGamal(ELGAMAL, 1985):

1. Definição dos parâmetros do ElGamal: assuma que p seja um número primo grande e q um número primo menor que p tal que q $|(p$ - 1). Seja $g \in \mathbb{Z}_p^*$ um elemento de ordem q e G o grupo multiplicativo de $\mathbb{Z}_p^*$ gerado por g. Seja x um número aleatório $\in \mathbb{Z}_p^*$ e $y = g^x$. Os parâmetros p, g e y são a chave pública. O número x é a chave privada que será gerada no próximo passo;

2. Geração distribuída de chaves: a geração distribuída de chaves (GDC) proposta por PEDERSEN (1991), também chamada de (k, n)-threshold ou (k,

[51]Tradução do termo em inglês *correctness*.
[52]Tradução do termo em inglês *privacy*.
[53]Tradução do termo em inglês *robustness*.

n)-GDC, permite que um conjunto de n jogadores gere juntamente um segredo de forma que o mesmo fique distribuído entre os jogadores sendo necessário um subgrupo de tamanho maior ou igual a k para revelar, enquanto um subgrupo menor que k não consegue obter qualquer informação. Os jogadores geram o par de chave pública e privada do ElGamal, onde ao final todos os jogadores conhecem a chave pública e compartilham um segredo x_i da chave privada x sendo $\sum_{i=1}^{k} x_i = x \bmod q$. O protocolo de PEDERSEN (1991) é descrito na Seção 2.3.8;

3. Geração da chave simétrica: cada par de jogadores (P_i, P_j) acorda uma chave secreta $k_{i,j}$ para ser usada em uma criptografia simétrica, o que torna possível uma comunicação segura entre os jogadores;

4. Representação das cartas em memória: cada jogador calcula antecipadamente os valores das 52 cartas como $g^0, g^1, ..., g^{51} \in G$;

5. Cálculo do grupo de combinação de cartas: assumindo que k é o número de jogadores e que o baralho contém 52 cartas, cada jogador calcula e armazena as seguintes mensagens criptografadas: $D_i = E(g^{52i})$ para i = 0,..., k - 1, onde $S = \{D_0, ..., D_{k-1}\}$. Como será visto a seguir, o grupo S é utilizado para verificar se uma carta já foi gerada anteriormente;

6. Comprometimento com a carta: cada jogador P_i escolhe $r_i \leftarrow \{0, ..., 51\}$, calcula $C_i = E(g^{ri})$ e anuncia um comprometimento de C_i por meio de uma função *hash*. Se um jogador tentar trapacear ao escolher $r_i \notin \{0, ..., 51\}$, nenhum problema ocorre no protocolo;

7. Provando o comprometimento: cada jogador P_i revela C_i e todos os jogadores verificam se todos os valores comprometidos no passo anterior estão corretos. Se um ou mais valores comprometidos estão errados, o jogador que trapaceou é banido e um novo grupo é formado com os outros jogadores;

8. Gerando carta criptografada: usando a propriedade de criptografia homomórfica aditiva do ElGamal, os jogadores calculam $E(\prod_i g^{ri}) = E(g^c)$, onde $c = \sum_i r_i \bmod q$. Se todos os jogadores escolherem honestamente os valores de r, $c \in \{1, ..., 51k\}$;

9. Testando colisão: sendo $E(m_1)$ e $E(m_2)$ duas mensagens criptografadas com ElGamal, JAKOBSSON e SCHNORR (1999) propõem um teste de igualdade de texto[54] que a partir de $E(m_1)$ e $E(m_2)$ é possível verificar se $m_1 = m_2$ sem revelar m_1 ou m_2 ou qualquer outra informação. Em seu protocolo, dado o

[54]Tradução da expressão em inglês *oblivious test of plaintext equality*.

conjunto $(g,\ y,\ m,\ s)$ e x como chave privada, é necessário determinar e provar se $\log_g(y) = \log_m(s)$. O protocolo tem a seguinte forma:

(a) Configuração: seleciona-se um número aleatório $a \in \mathbb{Z}_p^*$;

(b) Prova de primeira ordem: seja o conjunto $(\bar{s},\ \bar{\sigma},\ \bar{m}) = (s^a, m^{ax}, m^a)$. Verifica-se se $\bar{s} = \bar{\sigma}$;

(c) Prova de segunda ordem: Verifica-se se $\log_m(\bar{m}) = \log_s(\bar{s})$ e também se $\log_g(y) = \log_{\bar{m}}(\bar{\sigma})$.

Como forma de verificar e exemplificar o uso do protocolo acima para determinar se duas cartas são iguais, assuma duas cartas: c_1 e c_2. As duas cartas criptografadas possuem a seguinte forma: $E(c_1) = E(g^{c_1}) = (g^{r_1},\ g^{c_1}y^{r_1}) = (a_1,\ b_1)$ e $E(g^{c_2}) = (g^{r_2},\ g^{c_2}y^{r_2}) = (a_2,\ b_2)$, sendo r_1 e r_2 números aleatórios e $y = g^x$.

Utilizando o homomorfismo multiplicativo do ElGamal, calcula-se $E(c_1)/E(c_2) = (g^{r_1-r_2}, g^{c_1-c_2}y^{r_1-r_2})$. Para testar se duas cartas são iguais, o problema é equivalente a testar se uma mensagem criptografada é o número 1 criptografado. Portanto, se $c_1 = c_2$, então $E(c_1)/E(c_2) = (g^{r_1-r_2}, g^0 y^{r_1-r_2}) = (g^r, 1 * y^r) = (g^r, y^r) = (a,\ b)$.

Utilizando o protocolo de JAKOBSSON e SCHNORR (1999), o conjunto $(g,\ y,\ m,\ s)$ é substituído por $(g,\ y,\ a,\ b)$ para provar se $\log_g(y) = \log_a(b)$.

Para testar a prova de primeira ordem, o conjunto $(\bar{b},\ \bar{\sigma},\ \bar{a}) = (b^{r'}, a^{r'x}, a^{r'})$ e r' é um número aleatório. Como o protocolo de PEDERSEN (1991) distribui a chave privada entre os jogadores, ou seja, $\sum_{i=1}^{k} x_i = x$, o fator a^x pode ser calculado por todos os jogadores de forma que $a^x = \prod_{i=1}^{k} a^{x_i}$. Portanto, é necessária a participação de todos os jogadores para executar o protocolo de igualdade de cartas. A prova de primeira ordem pode ser verificada da seguinte forma:

$$\bar{b} = \bar{\sigma}$$

$$b^{r'} = a^{r'x}$$

$$y^{rr'} = g^{rr'x}$$

$$g^{rr'x} = g^{rr'x}$$

A primeira parte da prova de segunda ordem pode ser verificada da seguinte forma:

$$\log_a(\bar{a}) = \log_b(\bar{b})$$

$$\log_{g^r}(g^{rr'}) = \log_{y^r}(y^{rr'})$$

$$r' = r'$$

Por fim, a segunda parte da prova de segunda ordem pode ser verificada da seguinte forma:

$$\log_g(y) = \log_{\bar{a}}(\bar{\sigma})$$

$$\log_g(y) = \log_{g^{rr'}}(g^{rr'x})$$

$$\log_g(y) = x$$

$$y = g^x$$

Como parte do processo de verificação de igualdade de cartas, os jogadores mantêm uma lista $L = \{E(c_1, ..., E(c_t))\}$ contendo todas as cartas criptografadas que já foram distribuídas durante o jogo. Essa lista mantém cartas que foram geradas de forma aberta e também cartas que foram geradas de forma fechada. Se $L \neq 0$, os jogadores devem testar se uma nova carta gerada $E(g^c) \in L$. Para cada carta criptografada $E(g^{c'}) \in L$, os jogadores devem determinar se $c = c' \mod 52$ da seguinte forma:

(a) Usando o homomorfismo do ElGamal, os jogadores calculam $E(g^c/g^{c'}) = E(g^{c-c'})$;

(b) Utilizando o teste de igualdade de texto proposto por JAKOBSSON e SCHNORR (1999), os jogadores comparam $E(g^{c-c'})$ com cada valor criptografado do conjunto S. Se alguma igualdade é encontrada, a carta c já saiu anteriormente. O protocolo aborta e os jogadores iniciam novamente o protocolo de geração de carta.

Se não foi encontrada nenhuma igualdade, ou seja, se para cada $E(g^{c'}) \in L$, $E(g^{c-c'}) \notin S$, a carta c ainda não foi distribuída no jogo;

10. Gerando uma carta aberta: Cada jogador revela r_i e o número aleatório usado para gerar $E(g^{ri})$. Os jogadores verificam se todos $r_i \in \{0, ..., 51\}$ e se os valores criptografados estão corretos. Se um ou mais jogadores trapacearam, o protocolo aborta e um novo grupo é formado pelos outros jogadores. Se todos os jogadores foram honestos, a carta gerada é $\Sigma_i r^i \mod 52$;

11. Gerando uma carta fechada: Cada jogador revela somente para P_j o valor de r_i e o número aleatório usado para gerar $E(g^{ri})$. Nesse momento utiliza-se a chave secreta acordada entre o par P_i, P_j no passo 3 para estabelecer uma

comunicação segura com criptografia simétrica. O jogador P_j verifica se todos $r_i \in \{0, ..., 51\}$ e se os valores foram criptografados corretamente. Se um ou mais jogadores trapacearam, o protocolo aborta e um novo grupo é formado pelos outros jogadores. Se todos os jogadores foram honestos, a carta gerada é $\Sigma_i r^i$ mod 52.

O passo 2 garante que nenhum jogador consiga descriptografar a carta criptografada, pois somente com a colaboração de todos os jogadores é possível reconstruir a chave privada. De fato o protocolo não precisa em nenhum momento reconstruir a chave privada.

No passo 9 os jogadores tem o objetivo de verificar se uma carta $E(g^c)$ já foi gerada anteriormente. Essa verificação é feita com a carta criptografada e sem revelar o valor da carta por meio do teste de igualdade de texto. Para cada carta criptografada $E(g^{c'}) \in L$, os jogadores computam $E(g^{c-c'})$. No entanto, como chegar matematicamente na expressão $E(g^{c-c'})$? Para isso, lembrando que na criptografia ElGamal a mensagem m criptografada é composta por duas partes (a, b), e seja $E(g^c) = (a_1, b_1)$ e $E(g^{c'}) = (a_2, b_2)$, a expressão $E(g^{c-c'})$ é equivalente a $E(g^c)/E(g^{c'})$ $= (a_1 a_2^{-1}, b_1 b_2^{-1})$.

Para um melhor entendimento da motivação do uso da propriedade homomórfica multiplicativa para verificar se uma carta já foi gerada e também do uso do conjunto S definido no passo 5, a seguir encontra-se uma breve argumentação. Considere um exemplo composto por 2 jogadores, P_1 e P_2, e um baralho de 4 cartas apenas. As cartas são identificadas pelos números 0, 1, 2 e 3. Na primeira rodada a carta 2 foi gerada. Na segunda rodada, o jogador P_1 escolhe o número 3 e o jogador P_2 também escolhe o número 3. O valor da carta gerada é $3 + 3 = 6$. Aparentemente $c \neq c'$, pois $2 \neq 6$. No entanto, as cartas 2 e 6 são a mesma carta, pois a carta 6 deve ser reduzida pelo módulo do número total de cartas, ou seja, $6 \equiv 2 \bmod 4$. Como solução desse problema, o conjunto S é gerado no passo 5. Nesse exemplo, $S = \{E(g^{4*0}), E(g^{4*1})\}$ ou $S = \{E(g^0), E(g^4)\}$. Dessa forma, utilizando o homomorfismo multiplicativo, calcula-se $E(g^{6-2}) = E(g^4)$. Utilizando o teste da igualdade de texto descrito no passo 9, o valor $E(g^4)$ é comparado com cada elemento de S. Portanto, nesse exemplo verifica-se que $E(g^4) \in S$, ou seja, a carta já foi gerada em uma rodada anterior.

Capítulo 3

Proposta de Biblioteca de Suporte a Ações em Jogos Ponto-a-Ponto

Este capítulo apresenta a proposta conceitual da biblioteca de suporte a determinadas ações em jogos ponto-a-ponto, como ações relacionadas à sorte com dados e cartas. Para tanto, é apresentado a estrutura básica que compõe o protocolo de comunicação da biblioteca. Também são apresentados os dois módulos que compõem a biblioteca chamados de *Dice Rolling* e *Mental Poker*, sendo o primeiro para jogar dados e o segundo para jogar cartas. Para cada módulo é realizada uma proposta de protocolo criptográfico ou operações, a modelagem conceitual, como cada módulo funciona internamente e também como usá-lo. Por fim são descritas as decisões arquiteturais adotadas na modelagem da biblioteca.

3.1 Base de comunicação dos protocolos

Um protocolo criptográfico envolve a troca de mensagens entre os participantes a fim de realizar uma tarefa. Em termos de processo de comunicação, deve existir alguma estrutura que represente e permita essa troca de mensagens. Com esse objetivo, a Figura 3.1 apresenta o diagrama de classes que tem por responsabilidade permitir a comunicação entre os jogadores participantes do protocolo.

Os módulos que compõem a biblioteca e que são descritos neste capítulo utilizam essa estrutura básica como forma de comunicação entre os jogadores do protocolo. Uma mensagem trocada entre os jogadores no protocolo é representada pela classe `Message`. A classe `Message`, como o o próprio nome indica, tem a responsabilidade de carregar informações - mensagem - entre os jogadores. Sua estrutura é composta pelos campos *Method*, *Headers* e *Data*. O campo *Method* é um *enumerator*[1] do tipo `MessageMethod` que indica qual ação, ou passo, o protocolo está executando.

[1] *Enumerator*, também chamado de *enum*, é um tipo de dado composto por um conjunto de elementos nomeados que atuam como identificadores e constantes.

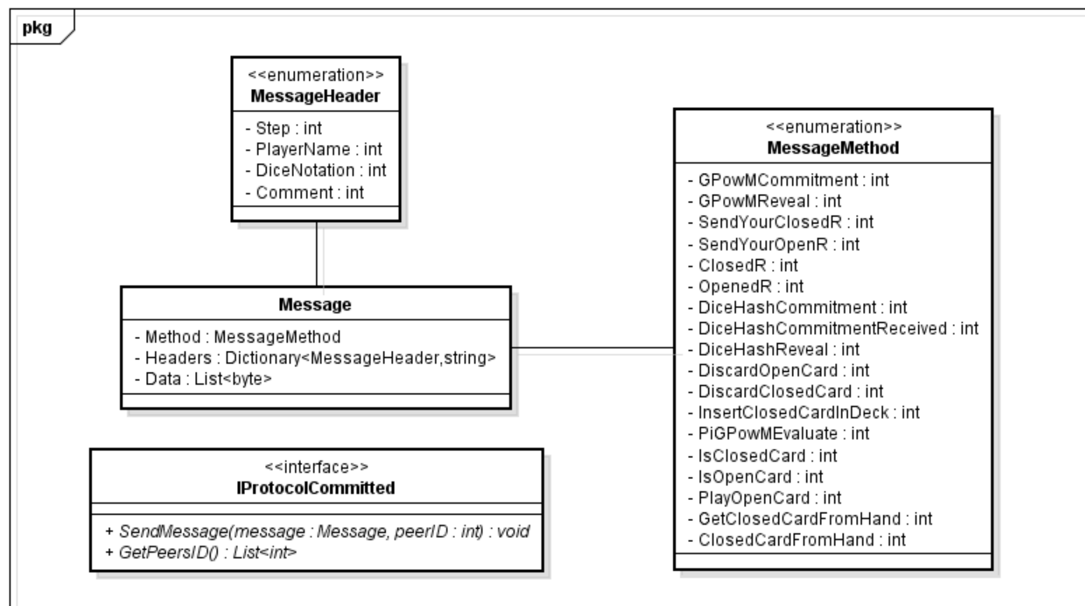

Figura 3.1: Diagrama de classes que compõem a base da comunicação dos protocolos

Esse campo é obrigatório, pois não existe mensagem sem uma ação. Nesse caso, seria uma mensagem vazia. Os tipos de métodos de `MessageMethod` serão descritos ainda nesse capítulo à medida que a arquitetura de cada módulo for descrita. Como um exemplo, o método do tipo *FirstDiceHashCommitment* utilizado no módulo *Dice Rolling* indica que um jogador está enviando o seu valor comprometido de um dado. O campo *Headers* é um *enumerator* do tipo `MessageHeader` que tem a responsabilidade de carregar informações extras que podem fazer sentido num determinado momento. Esse campo é opcional. Como exemplo, o *header* do tipo *Step* indica a rodada em que um jogo se encontra, e o campo *Comment* pode informar um comentário feito por algum jogador naquela jogada. A classe `Message` pode carregar zero ou mais `MessageHeader`. Por fim, o campo *Data* carrega a mensagem composta por um *array* de *bytes*. De fato, o campo *Data* armazena os valores numéricos e criptografados em *bytes* que são trocados e utilizados no protocolo criptográfico.

A interface `IProtocolCommitted` é obrigatória para qualquer classe que utilize os protocolos ciptográficos da biblioteca. A implementação dessa interface significa que uma classe é capaz de enviar uma mensagem para um determinado jogador ou *peer* via chamada do método `SendMessage(message:Message, peerID:int):void`. Além disso é possível obter uma lista com o identificador de cada *peer* participante do protocolo via chamada do método GetPeersID():List<int>. Portanto, a classe que estiver utilizando algum módulo da biblioteca será capaz de enviar uma mensagem do tipo `Message` para qualquer *peer*, pois o próprio módulo da biblioteca que estiver sendo utilizado chamará internamente esse método para enviar uma

mensagem do protocolo.

3.2 Módulo *Dice Rolling*

Esta subseção apresenta o primeiro módulo da biblioteca chamado de *Dice Rolling* que provê a capacidade de jogar dados em uma forma distribuída entre um conjunto de jogadores.

3.2.1 Proposta de protocolo para n jogadores

O protocolo de jogar moeda, que também pode ser utilizado para jogar dado, descrito na seção 2.3.9 descreve a interação de apenas dois jogadores. A seguir é proposto uma extensão desse protocolo com a participanção de n jogadores:

1. O jogador P_1 gera um número aleatório R_1 de 64 bits. Por meio de uma função *hash* SHA, P_1 calcula o valor *hash* de R_1 e envia para $P_2, ..., P_n$;

   $\text{SHA}(R_1)$

2. Como o jogador P_1, os jogadores $P_2, ..., P_n$ também executam o passo anterior, onde cada jogador se compromete com o seu número aleatório;

   $\text{SHA}(R_2)$, ..., $\text{SHA}(R_n)$

3. P_1 envia para $P_2, ..., P_n$ o seu número aleatório R_1;

   R_1

4. Como o jogador P_1, os jogadores $P_2, ..., P_n$ também executam o passo anterior, onde cada jogador envia seu número aleatório para os outros jogadores;

   R_2, ..., R_n

5. Nesse momento, cada jogador compara o número *hash* e o número aleatório que recebeu de cada jogador;

   $\text{SHA}(R_1^{'}) = \text{SHA}(R_1)$, ..., $\text{SHA}(R_n^{'}) = \text{SHA}(R_n)$, onde $R_n^{'}$ é o número comprometido ao final do passo 2 e R_n é o número aleatório enviado ao final do passo 4;

6. Cada jogador calcula o resultado da seguinte forma: $(R_1 + ... + R_n)$ *mod* 6.

Como instância do protocolo acima, a seguir é descrito o protocolo com três jogadores:

1. Alice gera um número aleatório R_A de 64 bits. Por meio de uma função *hash* SHA, Alice calcula o valor *hash* de R_A e envia para Bob e Carol;

   SHA(R_A)

2. Bob gera um número aleatório R_B de 64 bits . Por meio de uma função *hash* SHA, Bob calcula o valor *hash* de R_B e envia para Alice e Carol;

   SHA(R_B)

3. Carol gera um número aleatório R_C de 64 bits . Por meio de uma função *hash* SHA, Carol calcula o valor *hash* de R_C e envia para Alice e Bob;

   SHA(R_C)

4. Alice envia para Bob e Carol o seu número aleatório R_A;

   R_A

5. Bob e Carol confirmam se o número que Alice se comprometeu no passo 1 é igual ao valor *hash* do número enviado no passo 4.

   SHA(R_A) = SHA($R_A^{'}$), onde SHA(R_A) é o comprometimento enviado no passo 1 e $R_A^{'}$ é o número de 64 bits enviado no passo 4

6. Bob envia para Alice e Carol o seu número aleatório R_B;

   R_B

7. Alice e Carol confirmam se o número que Bob se comprometeu no passo 2 é igual ao valor *hash* do número enviado no passo 6.

   SHA(R_B) = SHA($R_B^{'}$), onde SHA(R_B) é o comprometimento enviado no passo 2 e $R_B^{'}$ é o número de 64 bits enviado no passo 6

8. Carol envia para Alice e Bob o seu número aleatório R_C;

   R_C

9. Alice e Bob confirmam se o número que Carol se comprometeu no passo 3 é igual ao valor *hash* do número enviado no passo 8.

   SHA(R_C) = SHA($R_C^{'}$), onde SHA(R_C) é o comprometimento enviado no passo 3 e $R_C^{'}$ é o número de 64 bits enviado no passo 8

10. Alice, Bob e Carol confirmam entre eles se o resultado é válido;

11. Alice, Bob e Carol calculam o resultado da seguinte forma: $(R_A + R_B + R_C)$ *mod* 6.

Ao final do passo 3, todos os jogadores estão comprometidos com os seus números, pois cada um enviou o seu número *hash* e também recebeu o número *hash* dos outros jogadores. Dessa forma, qualquer tipo de trapaça é descoberta, pois cada participante verifica se o número *hash* comprometido pelo outro jogador é igual ao número *hash* gerado a partir do número original enviado pelo outro jogador.

O protocolo proposto também evita qualquer tipo de trapaça caso dois jogadores se juntem para jogar contra o jogador restante. Supondo que Alice e Bob trapaceiem escolhendo os números entre eles no passo 1 e 2, o número resultante ainda seria aleatório, pois o número gerado por Carol no passo 3 ainda seria aleatório. Mais ainda, caso Alice e Bob alterem os seus números originais, Carol descobriria a trapaça conferindo o número *hash* comprometido por Alice e Bob nos passos 1 e 2.

O protocolo proposto gera 12 trocas de mensagens entre os três jogadores. A medida que o número de jogadores aumenta, o número de mensagens aumenta. Como mostra a Figura 3.2, temos 24 trocas de mensagens com 4 jogadores, 40 trocas de mensagens com 5 jogadores e 60 trocas de mensagens com 6 jogadores.

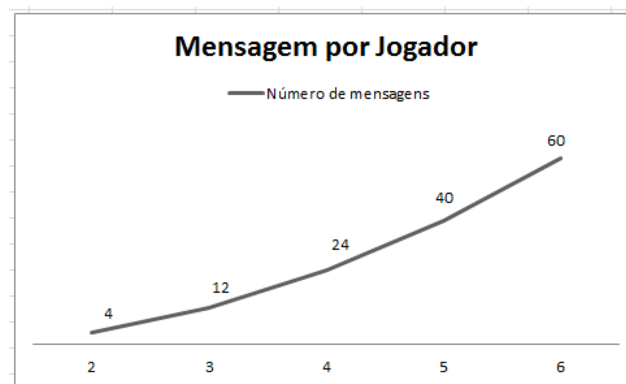

Figura 3.2: Número de mensagens por número de jogadores

Seja a variável n o número de jogadores participantes do protocolo, cada jogador deve enviar para os outros (n - 1) jogadores 2 mensagens, uma contendo o valor *hash* de comprometimento e outra contendo o valor aleatório. Logo, o número total de mensagens é $2 * n * (n - 1) = 2n^2 - 2n$. Portanto, a complexidade é $O(n^2)$[2] para a operação de jogar dados.

Caso o número de jogadores aumente consideravelmente, o sistema composto por esses *peers* pode ficar comprometido devido a grande quantidade de troca de mensagens, podendo, de fato, aumentar o tempo para gerar o número do dado. Além disso, caso seja necessário jogar, por exemplo, três dados, o número de mensagens triplica. Por exemplo, supondo um sistema com 5 jogadores, onde um jogador tenha

[2]A notação *Big O* é uma notação matemática aplicada para analisar o desempenho e comportamento de um algoritmo quando os dados de entrada crescem para valores muito grandes que seguem para o infinito.

que jogar três dados, o número de mensagens trocadas entre esses jogadores é três vezes 40 mensagens, ou 120 mensagens.

Fica claro que um sistema composto por um número grande de jogadores poderá ficar comprometido, pois o tempo para gerar um número de dado pode ser muito grande. O módulo *Dice Rolling* proposto a seguir implementa o protocolo para n jogadores descrito acima. Esse módulo também implementa a estratégia de executar o protocolo n vezes para jogar n dados. Além do mais, como alternativa, o módulo implementa um outro modo de jogar os dados, o chamado modo em lote. Por exemplo, para jogar 4 dados, em vez de executar o protocolo quatro vezes, uma vez para cada dado, o protocolo é executado apenas uma vez, gerando os quatro números em apenas uma rodada.

3.2.2 Notação padrão de dados

No módulo *Dice Rolling*, o jogador expressa a sua jogada por meio de uma notação de dado. A notação de dado é um sistema que serve para representar diferentes combinações de dados, geralmente em jogos de *role-playing game*, usando uma notação como 3d10 + 4, que representa o jogar de três dados de dez faces, com o resultado sendo a soma dos valores desses três dados mais o número quatro.

Na notação padrão, o caractere **d** representa o dado. O número que precede a letra **d** indica o número de dados. O número que aparece depois da letra **d** indica o número de faces do dado. A notação pode ser modificada com uma operação matemática e um número (KIPM, 2013):

XdY [-][+][x][/] N

X dados com Y faces modificados por meio de uma operação matemática com o número N. Um segundo dado também pode ser usado em vez de um número (KIPM, 2013):

XdY [-][+][x][/] AdB

X dados com Y faces modificados por meio de uma operação matemática com A dados de B faces.

Ainda como uma outra variação comum, é possível escrever a notação na forma XdY - [L][H], onde o caractere **L** significa o dado com o menor valor e o caractere **H** o dado com o maior valor. Por exemplo, 3d6 - L significa jogar três dados de seis faces e eliminar o dado de menor valor. Em 4d6 - H, quatro dados de seis faces são jogados e o maior dado é eliminado.

A notação padrão é uma evolução de outra notação introduzida em 1974, com a edição "*Blue Box*" do jogo *Dungeons & Dragons*[3] (WINTER, 2007). Na notação original de *Dungeons & Dragons*, os dados eram indicados pela amplitude de resultados. Por exemplo, 2d6 era expressado como 2-12, 3d10 era 3-30 (WINTER, 2007). A notação que indica a amplitude de resultados tem uma vantagem por tornar imediato a identificação dos possíveis valores. A Tabela 3.1 apresenta alguns valores representados na notação de amplitude de resultados e também na notação padrão. Com a notação de amplitude de resultados é imediato identificar qual é a melhor opção para um ataque ao inimigo. Nesse exemplo, é fácil identificar que a opção 2-16 oferece um valor maior de dano ao inimigo.

Tabela 3.1: Comparação entre os tipos de notação de dados (adaptado de WINTER (2007))

Notação padrão	Notação de amplitude
1d8	1-8
2d6	2-12
2d4 + 1	3-9
2d8	2-16
1d10	1-10

No entanto, a notação de amplitude de resultados tem algumas desvantagens. Identificar a combinação de dados que deve ser usada para gerar uma determinada amplitude de resultados pode ser uma tarefa desafiadora. Por exemplo, para gerar 3-12, deve-se usar 3d4 ou 1d10 + 2? Para gerar 30-300, deve-se usar 30d10, 1d10 x 30, 5d10 x 6? A probabilidade de resultados muda significativamente dependendo de qual estratégia usar. Com a notação padrão, esses problemas são evitados. Como será discutido ainda nesse capítulo, o módulo *Dice Rolling* implementa uma versão da notação padrão, e também oferece a possibilidade para o usuário da biblioteca implementar a sua própria versão da notação padrão.

3.2.3 Modelagem

Para modelar o módulo *Dice Rolling* foram necessários alguns ciclos de concepção, implementação, teste, validação e aperfeiçoamento para que o módulo pudesse ficar simples de usar, que atendesse a um objetivo claro, o de jogar dado de forma segura entre dois ou mais jogadores, e que também oferecesse a possibilidade do usuário estender determinados comportamentos. A Figura 3.3 representa o diagrama de classes a nível de projeto do módulo *Dice Rolling*.

[3] *Dungeons & Dragons* é um *role-playing game*, ou *RPG*, de fantasia medieval, publicado pela primeira vez em 1974. É considerado como a origem dos *RPG* modernos.

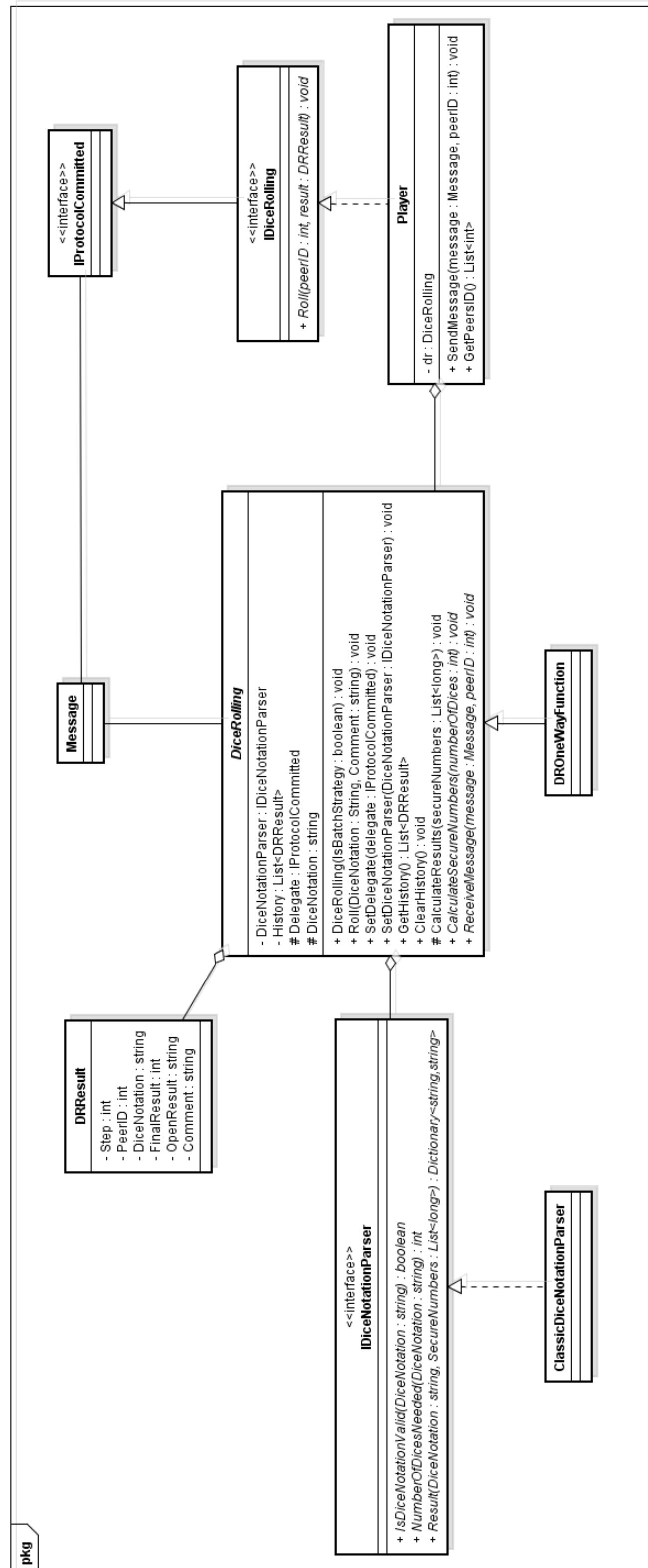

Figura 3.3: Diagrama de classes de projeto do módulo *Dice Rolling*

A classe central do módulo é a classe abstrata `DiceRolling`, que provê a orquestração de todas as outras classes, além de servir como o ponto de acesso ao módulo. A classe `DiceRolling` tem a responsabilidade de receber uma notação de dado e retornar um resultado, além de oferecer um histórico de todas as jogadas anteriores. Por ser uma classe abstrata, ela não pode ser instanciada, mas sim implementada por meio de dois métodos abstratos: `CalculateSecureNumbers(numberOfDices int): void` e `ReceiveMessage(message: DRMessage, peerID: int): void`. O primeiro método tem a função de calcular, por meio da implementação do protocolo criptográfico *Dice Rolling* descrito anteriormente, `numberOfDices` números que serão utilizados na avaliação da notação de dado. O segundo método tem a responsabilidade de receber cada mensagem enviada por outro *peer* a fim de permitir a troca de mensagens necessárias para a implementação do procolo.

Como descrito na Seção 3.1 a classe `Message` é a unidade de mensagem que é trocada entre os *peers* a cada passo do protocolo. Essa classe contém a mensagem enviada de um *peer* para o outro, o passo corrente em que o protocolo se encontra, a notação de dado da jogada e também um comentário que indica o contexto da jogada, como "matando um monstro".

Como descrito na Seção 3.1 a interface `IProtocolCommitted` permite que o usuário use esse módulo por exigir a implementação de dois métodos abstratos: `SendMessage(message: Message, peerID: int): void` e `GetPeersID(): List<int>`. O módulo *Dice Rolling* não sabe enviar uma mensagem, pois depende de qual arquitetura ou tecnologia está sendo usada. Portanto, quem sabe enviar uma mensagem é o próprio usuário do módulo. Dessa forma, o primeiro método abstrato exige que o usuário tenha a responsabilidade de enviar uma mensagem para outro `peer`. O segundo método retorna um identificador de cada *peer* que está participando do protocolo.

A interface `IDiceRolling` provê para a classe usuária do módulo *Dice Rolling* um ponto de acesso aos resultados das jogadas. O método `Roll(peerID: int, result: DRResult): void` é chamado internamente pelo módulo *Dice Rolling* passando o resultado da jogada via objeto `DRResult` e também o identificador do *peer* que jogou os dados. Portanto, sempre que um jogador iniciar uma jogada, todos os jogadores, inclusive o jogador que iniciou a jogada, receberão os resultados por meio desse método.

A interface `IDiceNotationParser` tem a responsabilidade de calcular o resultado do protocolo a partir de uma notação de dado. O método abstrato `Result(DiceNotation: String, SecureNumbers: List<long>): Dictionary<String, String>` recebe uma notação de dado e uma lista de números que foram gerados por meio do protocolo criptográfico, e retorna um dicionário com o resultado em forma de resultado aberto e resultado final. Por

exemplo, supondo uma notação de dado 1d6 + 2d4, e uma lista de números seguros 245, 25, 385 gerados pelo protocolo, o resultado final será 5 + 1 + 1 que é igual a 7, e o resultado aberto será a lista com os valores dos dados na forma (5, 1, 1). Dessa forma, é obtido o resultado final, assim como os valores individuais dos dados que determinaram o resultado final. Por fim, essa interface ainda exige que o usuário implemente mais dois métodos: `IsDiceNotationValid(DiceNotation> String)` e `NumberOfDicesNeeded(DiceNotation: String)`. O primeiro indica se uma notação de dado é válida, ou seja, se todos os operadores são válidos, se a ordem em que os termos aparecem estão corretos, etc. O segundo método analisa uma notação de dado e retorna a quantidade de dados necessários para calcular essa notação de dado. Por exemplo, com a notação 2d12 + 3d4 são necessários 5 dados.

Depois de gerar o resultado, a classe `DiceRolling` guarda um histórico das jogadas por meio de objetos `DRResult`. Pode-se recuperar, para uma determinada rodada, o nome do jogador que iniciou a jogada por meio de um número que identifica o *peer*, a notação de dado, o comentário da jogada, assim como o resultado final e o resultado aberto. O método `GetHistory(): List<DRResult>` retorna uma lista de objetos `DRResult` permitindo ao usuário manipular o histórico de jogadas por meio de operações em uma lista. O método `ClearHistory(): void` permite ao usuário apagar o histórico das jogadas caso necessário. A adição de um novo objeto `DRResult`, ou seja, o registro de uma nova jogada é feita somente pelo módulo *Dice Rolling*, e não pelo usuário do módulo.

3.2.4 Funcionamento interno

O módulo *Dice Rolling* é composto por um conjunto de classes, cada uma com responsabilidade bem definida. No diagrama de sequência da Figura 3.4, a ênfase está na ordem temporal das mensagens trocadas entre os objetos dessas classes a fim de executar as tarefas do módulo *Dice Rolling*.

A operação de jogar um dado é iniciada a partir do recebimento da mensagem assíncrona `Roll`, que envia uma notação de dado para ser calculada pelo módulo. Antes que o protocolo criptográfico seja iniciado, uma mensagem síncrona é enviada ao objeto `ClassicDiceNotationParser` para verificar se essa notação de dado é válida. Em caso positivo, outra mensagem síncrona é enviada para determinar a quantidade de dados que essa notação de dado exige. Em seguida, inicia-se a execução do protocolo criptográfico a partir da mensagem assíncrona `CalculateSecureNumbers`. Nesse momento os *peers* participantes trocam uma série de mensagens que tem por objetivo executar o protocolo criptográfico descrito na seção 3.2.1. O resultado do protocolo criptográfico é um conjunto de números de 64 *bits* utilizados para o cálculo dos valores dos dados. Dessa forma,

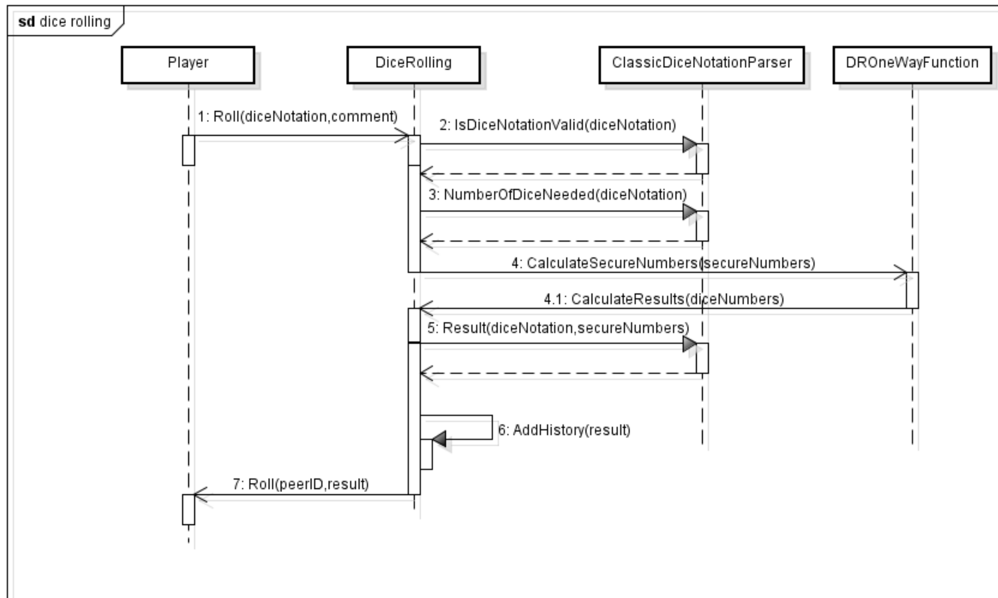

Figura 3.4: Diagrama de sequência das mensagens internas do módulo *Dice Rolling* na geração de dados

quando o protocolo criptográfico termina de executar, uma mensagem assíncrona é enviada via `CalculateResults` para a classe principal `DiceRolling` para cálculo da notação de dados. A partir disso, por meio da mensagem `Result`, a classe `ClassicDiceNotationParser` tem a função de calcular o resultado fechado e o resultado aberto da notação de dado com os números seguros gerados pelo protocolo criptográfico. Em seguida um novo registro é adicionado ao histórico de jogadas por meio da chamada `AddHistory` que adiciona o identificador do *peer*, o número da rodada, o resultado fechado, o resultado aberto e o comentário feito pelo jogador naquela rodada. Por fim, a execução termina quando o módulo *Dice Rolling* chama o método de *callback* `Roll(peerID:int, result:DRResult):void` da interface `IDiceRolling` passando o resultado daquela jogada e o identificador do *peer* que jogou.

3.2.5 Como usar o módulo

A seguir é apresentado um código fonte de exemplo do uso do módulo *Dice Rolling*. O código está escrito na linguagem de programação C#:

```
1   public class Player : IDiceRolling {
2       private DiceRolling _diceRolling;
3
4       public Player (DiceRolling dr) {
5           _diceRolling = dr;
6           _diceRolling.SetDelegate(this);
```

```
7         }
8
9         public void SendMessage(Message message, int peerID)
10         { /* implementação */ }
11
12        public List<int> GetPeersID()
13         { /* implementação */ }
19
20        public void SetDiceNotationParser
21                    (IDiceNotationParser diceNotationParser)
22        {
23             _diceRolling.SetDiceNotationParser(diceNotationParser);
24        }
25
26        public void Roll(int peerID, DRResult result)
27         { /* implementação */ }
28
29        public void RollDice(string diceNotation, string comment){
30             _diceRolling.Roll(diceNotation, comment);
31        }
32    }
```

Para a classe `Player` usar o módulo *Dice Rolling* é necessário implementar a interface `IDiceRolling` e fazer agregação com uma subclasse de `DiceRolling`. A interface `IProtocolCommitted` é implementada nas linhas 9 e 12, enviando mensagens para outros *peers* e informando o identificador de cada *peer* participante do protocolo. A interface `IDiceRolling` é implementada na linha 29, método onde o resultado de uma jogada será recebido. O construtor, na linha 4, recebe como parâmetro um tipo `DiceRolling`, o que permite um desacoplamento de dependência[4], permitindo o uso de diferentes implementações de protocolos criptográficos desde que sejam subclasses de `DiceRolling`. Na linha 6 é declarado a delegação, onde a classe do tipo `DiceRolling` irá delegar o envio de mensagens pela classe `Player`. Na linha 30 é realizado de fato a chamada do método `Roll`, passando a notação de dado e um comentário da jogada. A linha 23 mostra a possibilidade do usuário substituir o componente padrão que valida e calcula uma notação de dado.

Abaixo está um exemplo de execução do módulo *Dice Rolling* :

```
1    Player player = new Player(new DROneWayFunction(true));
```

[4]Esse padrão é conhecido como injeção de dependência, em inglês *dependency injection*, onde a dependência entre módulos não é definida programaticamente, mas sim pela “injeção” dessa dependência via polimorfismo.

```
2    player.RollDice("1d6 + 2d12 - 3", "Matando o monstro.");
```

Na linha 1 é feito a injeção de dependência com a classe concreta `DROneWayFunction`. O parâmetro `true` indica que o protocolo criptográfico será executado em lote, ou seja, o protocolo será executado apenas uma vez independente do número de dados (ver mais detalhes no final da seção 3.2.1). Na linha 2 o dado é jogado de fato, passando como argumentos a notação de dado e o comentário da jogada. O resultado dessa jogada será recebido na linha 26 na implementação da classe `Player`.

A seguir temos as formas de estender o módulo *Dice Rolling*. A primeira forma é definir outro protocolo criptográfico para jogar os dados.

```
1    public class MyOwnProtocol : DiceRolling {
2        public override void CalculateSecureNumbers
3                                        (int numberOfDices)
4        { /* implementação */ }
5
6        public override void ReceiveMessage(Message message, int peerID)
7        { /* implementação */ }
8    }
9
10    Player player = new Player(new MyOwnProtocol(false));
```

Depois de herdar da classe `DiceRolling`, a classe `MyOwnProtocol` poderá ser usada como implementação do protocolo criptográfico como mostrado na linha 10 acima. Nesse exemplo o protocolo não será executado em lote, ou seja, caso a notação tenha 5 dados, o protocolo será executado 5 vezes, uma para cada dado.

A outra forma de estender o módulo é definir a própria notação de dado que será utilizada para jogar os dados entre os *peers*:

```
1    public class MyOwnDiceNotationParser : IDiceNotationParser {
2        bool IsDiceNotationValid(string diceNotation)
3        { /* implementação */ }
4
5        int NumberOfDicesNeeded(string diceNotation)
6        { /* implementação */ }
7
8        Dictionary<string, string> Result
9                (string diceNotation, IList<long> secureNumbers)
10        { /* implementação */ }
11    }
```

```
12
13    player.SetDiceNotationParser(MyOwnDiceNotationParser);
```

A linha 13 do código acima mostra que a notação de dado padrão é substituída pela notação implementada por `MyOwnDiceNotationParser`.

Por fim, a Figura 3.5 exibe um diagrama de sequência como exemplo das possíveis interações de uma classe usuária do módulo *Dice Rolling*:

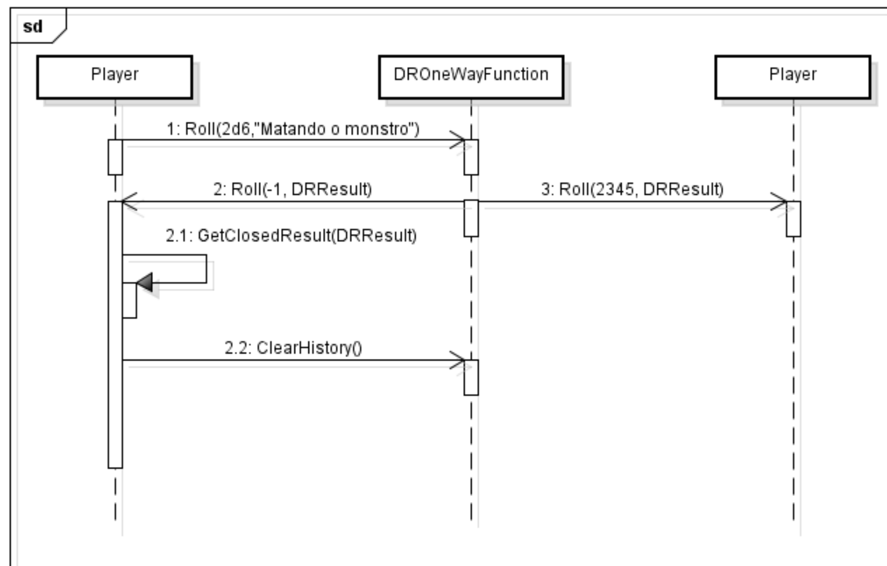

Figura 3.5: Diagrama de sequência de exemplo de uso do módulo *Dice Rolling*

A classe `Player` inicia com uma mensagem assíncrona `Roll` para jogar os dados passando como argumentos a notação de dado "2d6"e o comentário "Matando o monstro". Por sua vez, a classe `DROneWayFunction` executa o cálculo dos dados via protocolo criptográfico e retorna o resultado passando o identificador do jogador que iniciou a jogada e o resultado num objeto do tipo `DRResult`. Para o jogador que iniciou a jogada, o valor retornado para o identificador é -1, pois foi ele mesmo que jogou. Para o outro jogador, o identificador retornado tem valor 2345. Em seguida, o jogador realiza mais algumas operações como recuperar o resultado fechado da jogada por um método que ele implementou chamado `GetClosedResult`, retornando a propriedade `FinalResult` da classe `DRResult`. Por fim, o jogador decide apagar todo o histórico do jogo chamando o método `ClearHistory`.

3.3 Módulo *Mental Poker*

Esta subseção apresenta o segundo módulo da biblioteca chamado de *Mental Poker* que provê a capacidade de jogar cartas em uma forma distribuída entre um conjunto de jogadores.

3.3.1 Proposta de extensão

Como descrito na subseção *Uma abordagem orientada ao pôquer* da seção 2.3.10, GOLLE (2005) propõe um protocolo de *Mental Poker* que permite a distribuição de cartas entre os jogadores de forma segura. A carta gerada e distribuída pelo protocolo pode ser aberta, ou seja, visível entre todos os jogadores ou ser uma carta fechada, sendo visível apenas por um único jogador. A grande diferença do protocolo está no fato de que não é preciso embaralhar todo o baralho antes da primeira rodada, diminuindo o custo computacional e a latência de comunicação entre os jogadores. As cartas são geradas à medida que uma nova carta precisa ser distribuída. Esse protocolo é ideal para jogos que usam apenas uma parte do baralho e que não distribuem todas as cartas entre os jogadores de uma só vez, características de um jogo de pôquer.

Todavia, o protocolo proposto por GOLLE (2005) descreve como distribuir cartas entre jogadores, mas não descreve nenhum tipo de operação com cartas, como por exemplo jogar uma carta aberta na mesa ou descartar uma carta. Por isso, com o objetivo de estender o protocolo, o módulo *Mental Poker* proposto neste trabalho implementa e estende o protocolo de GOLLE (2005) com as seguintes operações:

- Jogadores geram uma carta para a mesa de forma aberta;
- Jogador joga carta de sua mão na mesa de forma aberta;
- Jogador coloca uma carta de sua mão novamente no baralho;
- Jogador P_i escolhe e pega carta da mão do jogador P_j;
- Jogador P_i recebe carta da mão do jogador P_j que foi escolhida pelo próprio jogador P_j;
- Jogador descarta carta de sua mão no monte de descarte de forma aberta;
- Jogador descarta carta de sua mão no monte de descarte de forma fechada;
- Jogadores incluem cartas do monte de descarte de novo no baralho.

Supondo que k seja o número de jogadores, para que seja possível implementar essas operações, cada jogador P_i mantém um dicionário que guarda todas as cartas criptografadas que cada jogador P_j para j = 1,...,k - 1 possui. Por exemplo, num jogo composto pelos jogadores Alice e Bob, Alice mantém um dicionário onde a chave é um identificador para o jogador Bob, e o valor é o conjunto das cartas criptografadas que Bob possui. Assuma que a mão (conjunto de cartas criptografadas) do jogador P_i seja o conjunto $H_i = \{E(c_1), ..., E(c_t)\}$. De maneira geral e estendendo o protocolo proposto por GOLLE (2005), a cada carta gerada para o jogador P_i, cada jogador P_j

para j = 1,...,k - 1 armazena que o jogador P_i agora possui essa carta ao adicionar a carta gerada ao conjunto H_i. Assuma que cada jogador mantém um conjunto S, sendo $D_i = E(g^{52i})$ para i = 0,..., k - 1, onde $S = \{D_0, ..., D_{k-1}\}$. Assuma também que T seja o conjunto que armazena todas as cartas que estão na mesa e que F seja o conjunto que armazena as cartas que estão no monte de cartas descartadas durante o jogo, também chamado de monte de descarte. Cada jogador mantém o estado dos conjuntos T e F durante a partida.

Para verificar se uma carta c pertence realmente ao jogador P_i, ou seja, se $c \in H_i$, deve-se seguir o algoritmo descrito a seguir. Para cada carta criptografada $E(g^{c'}) \in H_i$, os jogadores devem determinar se $c = c'$ mod 52 da seguinte forma:

1. Usando o homomorfismo do ElGamal, os jogadores calculam $E(g^c/g^{c'}) = E(g^{c-c'})$;

2. Utilizando o teste de igualdade de texto proposto por JAKOBSSON e SCHNORR (1999), os jogadores comparam $E(g^{c-c'})$ com cada valor criptografado do conjunto S. Se alguma igualdade for encontrada, a carta c de fato pertence ao jogador P_i. Lembrando que o protocolo de teste de igualdade de texto só pode ser realizado com a participação de todos os jogadores, pois a chave privada está distribuída entre eles.

Importante ressaltar que as operações propostas também mantêm as propriedades de correção, privacidade e robustez definidas no protocolo GOLLE (2005). A partir disso, as operações propostas são definidas abaixo:

- Jogadores geram uma carta para a mesa de forma aberta: por existir o conjunto T descrito acima que tem a função de armazenar todas as cartas que estão na mesa, os jogadores podem executar o mesmo protocolo de geração de uma carta aberta. No entanto, em vez de adicionar essa carta gerada em algum conjunto H_i que representa a mão de um jogador, os jogadores adicionam essa carta ao conjunto T.

- Jogador joga carta de sua mão na mesa de forma aberta: para implementar essa operação, deve ser possível garantir que a carta que o jogador P_i jogou de fato pertence a ele. Por exemplo, caso o jogador P_i possua em sua mão as cartas 3 e 5, e caso esse jogador jogue a carta de número 6 na mesa, como garantir que a carta 6 está criptografada em sua mão e que P_i não está trapaceando? Se o jogador P_i jogou na mesa a carta c, os demais jogadores devem verificar se essa carta $E(g^c) \in H_i$.

  Se não for encontrada nenhuma igualdade, ou seja, se para cada $E(g^{c'}) \in H_i$, $E(g^{c-c'}) \notin S$, a carta c não pertence ao jogador P_i. Os outros jogadores devem excluir o jogador P_i e fechar um novo grupo.

Se a carta c pertencer de fato ao jogador P_i, então a carta c é jogada na mesa e não pertence mais ao jogador P_i. Cada jogador também deve excluir $E(g^c)$ de H_i;

- Jogador coloca uma carta de sua mão novamente no baralho: no protocolo proposto por GOLLE (2005) o conjunto L armazena as cartas que já foram distribuídas durante o jogo. A carta c é armazenada criptografada no conjunto L, ou seja, na forma $E(g^c)$. A operação de um jogador P_i colocar uma carta c de sua mão novamente no baralho corresponde a remover $E(g^c)$ do conjunto L e cada jogador remover $E(g^c)$ de H_i. Portanto, a carta c poderá ser distribuída em outra rodada para qualquer outro jogador, inclusive para o jogador que inseriu novamente a carta no baralho. De fato quando a mesma carta for gerada posteriormente, não será exposta nenhuma informação de que é a mesma carta, pois a criptografia ElGamal é aleatória, ou seja, a mesma carta c pode ser representada por diferentes valores criptografados;

- Jogador P_i escolhe e pega carta da mão do jogador P_j: como o jogador P_i armazena o conjunto H_j, ou seja, as cartas que o jogador P_j possui em sua mão, P_i escolhe qualquer carta criptografada $E(g^c) \in H_j$ e anuncia para todos os outros jogadores que ele deseja pegar essa carta. No protocolo proposto por GOLLE (2005), cada par $P_{i,j}$ estabelece uma chave simétrica entre eles.

  O jogador P_j criptografa a carta c para formar $E(g^{c'})$ e envia para todos os outros jogadores. Por meio de um algoritmo de criptografia simétrica que utiliza uma chave secreta estabelecida entre os jogadores, o jogador P_j envia o valor da carta c e o fator aleatório r para P_i, sendo a carta c a carta criptografada em $E(g^c)$ escolhida por P_i e r é o número aleatório utilizado na geração de $E(g^{c'})$. Nesse momento o jogador P_i verifica, por meio de c e r, se o valor $E(g^{c'})$ gerado por P_j está correto. Para verificar se de fato a carta c enviada pelo jogador P_j para o jogador P_i corresponde a $E(g^c)$ escolhida por P_i, os jogadores calculam $E(g^c/g^{c'}) = E(g^{c-c'})$ e verificam por meio do teste de igualdade de texto se de fato $E(g^c) = E(g^{c'})$.

  Se o jogador P_j não trapaceou, todos os jogadores atualizam as cartas que os jogadores P_i e P_j possuem, sendo que agora a carta $E(g^c)$ foi removida de H_j e inserida em H_i. A carta c era conhecida apenas pelo jogador P_j. Agora a carta c é conhecida entre os jogadores P_i e P_j durante todo o jogo;

- Jogador P_i recebe carta da mão do jogador P_j que foi escolhida pelo próprio jogador P_j: operação semelhante à operação descrita acima. No entanto, agora é o jogador P_j quem escolhe $E(g^c) \in H_j$ e envia para o jogador P_i;

- Jogador descarta carta de sua mão no monte de descarte de forma aberta: jogador P_i escolhe uma carta $E(g^c)$ de sua mão H_i para jogar no conjunto de cartas descartadas F. O jogador P_i anuncia para todos os outros jogadores a carta c que $E(g^c)$ representa. Para garantir que a carta c corresponde a $E(g^c)$ anunciada pelo jogador P_i, todos os jogadores calculam $E(g^{c'})$ a partir da carta c anunciada por P_i. Calculando $E(g^c/g^{c'}) = E(g^{c-c'})$, os jogadores verificam por meio do teste de igualdade de texto se de fato $E(g^c) = E(g^{c'})$.

  Se $c = c$', cada jogador atualiza o conjunto H_i retirando $E(g^c)$. Por fim, $E(g^c)$ é inserida no conjunto F.

- Jogador descarta carta de sua mão no monte de descarte de forma fechada: jogador P_i escolhe uma carta $E(g^c)$ de sua mão H_i para jogar no conjunto F. O jogador P_i remove $E(g^c)$ de sua mão. Cada jogador também remove $E(g^c)$ de H_i. Por fim, $E(g^c)$ é inserida no conjunto F.

- Jogadores incluem cartas do monte de descarte de novo no baralho: para que seja possível gerar as cartas que outrora foram descartadas no monte de descarte, os jogadores devem remover os elementos do conjunto L para cada elemento presente no conjunto F. Além disso, todos os elementos do conjunto F também devem ser removidos, pois essas cartas passaram do monte de descarte para o baralho onde os jogadores pegam as cartas. Em suma, a interseção dos conjuntos L e F deve ser removida.

Além das operações propostas acima que estendem o protocolo de GOLLE (2005), este trabalho também propõe mais duas modificações no protocolo. A primeira modificação somente se aplica em jogos que utilizam cartas abertas. O objetivo desta modificação é eliminar a colisão no processo de geração de uma carta. Assumindo um baralho com 52 cartas, o protocolo de cartas gera um número entre 0 e 51. O usuário do protocolo deve manter uma tabela que mapeia esse número gerado pelo protocolo com o valor da carta do baralho que está sendo utilizado no jogo. Por exemplo, assumindo um baralho com 5 cartas, o usuário pode usar a Tabela 3.2 para o seu jogo.

Caso o número 1 seja gerado pelo protocolo na primeira rodada, então todos os jogadores sabem que a carta Dama de paus saiu, pois o jogo utiliza somente cartas abertas. Dessa forma, na segunda rodada, a carta Dama de paus não precisa mais ser gerada, pois caso seja gerada, acontecerá uma colisão e o protocolo deverá ser executado novamente. Portanto, para resolver este problema, em vez de gerar um número entre 0 e 4, o protocolo gera um número entre 0 e 3 na segunda rodada e a tabela de mapeamento deve ser atualizada, pois o intervalo de cartas mudou e

Tabela 3.2: Exemplo de mapeamento entre o número gerado pelo protocolo e as cartas do baralho

Número	Carta
0	Rei de ouros
1	Dama de paus
2	Vale de espadas
3	Ás de copas
4	Coringa

também o mapeamento entre o número e o valor da carta. Com esta proposta o conjunto L e S não são utilizados, pois não há colisão.

Esta primeira modificação tem a limitação de funcionar somente em jogos com todas as cartas abertas. Se existir uma carta fechada, a modificação não funciona. No entanto, a segunda proposta aproveita a seguinte característica comum em determinados jogos, como o jogo Escopa[5]: mesmo que o jogo contenha cartas fechadas, o jogo é desenvolvido em rodadas, sendo que ao fim de cada rodada todos os jogadores mostram todas as suas cartas jogando-as na mesa. Logo, a ideia é alterar o protocolo de geração de cartas para não gerar na rodada posterior as cartas que foram abertas na rodada anterior. Por exemplo, assuma novamente a Tabela 3.2 e um jogo com dois jogadores. Na primeira rodada, cada jogador recebe uma carta fechada. Como as cartas são fechadas, pode haver colisão no momento de gerar a segunda carta. Em seguida, a primeira rodada é finalizada quando cada jogador joga a sua carta na mesa, por exemplo, as cartas Rei de ouros e Dama de paus. Como essas duas cartas já saíram, a segunda rodada não precisa mais gerar essas cartas. Logo, no início da segunda rodada e antes de gerar qualquer carta, o mapeamento das cartas deve ser atualizado. Em seguida, o protocolo é alterado para gerar um número entre 0 e 2. Mais duas cartas fechadas são geradas, sendo que colisões podem ocorrer. O jogo continua nas rodadas seguintes com esta mesma ideia. Para por em prática esta modificação, além de atualizar o mapeamento durante as rodadas e atualizar o intervalo de número que o protocolo de cartas deve gerar, ao fim de cada rodada o conjunto L deve ser apagado para comportar as cartas que serão geradas (abertas ou fechadas) na rodada seguinte. O conjunto T que guarda as cartas que estão na mesa também deve ser apagado.

Como será visto no capítulo 5 que apresenta o experimento, essas modificações de fato reduzem a quantidade de *bytes* trafegados no protocolo, o tempo gasto em operações de criptografia e o tempo para gerar uma carta.

[5]Escopa é um jogo de cartas onde o objetivo é fazer o maior número de *escopas* que é a soma de 15 pontos com as cartas da mão do jogador e as cartas da mesa. O jogo Escopa utiliza um baralho de 40 cartas com no máximo 3 jogadores. Ao fim de cada rodada, todas as cartas são reveladas.

3.3.2 Modelagem

Muitos dos conceitos utilizados na modelagem do módulo *Dice Rolling* foram aproveitados para modelar o módulo *Mental Poker*. A Figura 3.6 representa o diagrama de classes a nível de projeto do módulo *Mental Poker*.

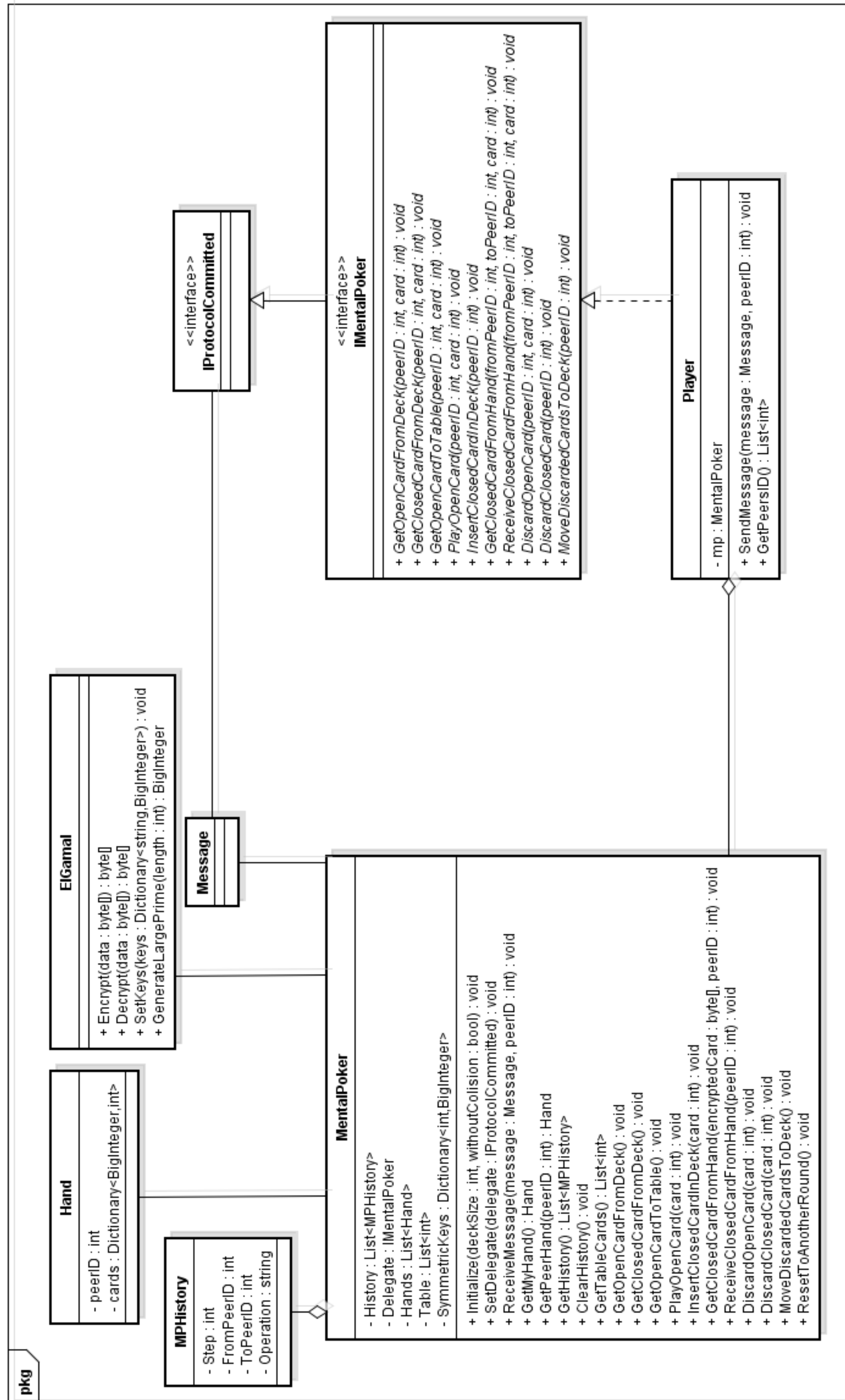

Figura 3.6: Diagrama de classes de projeto do módulo *Mental Poker*

A principal classe do módulo é a classe `MentalPoker`, pois é a partir dela que o usuário pode utilizar as operações do módulo de jogar cartas. O módulo é inicializado por meio do método `Initialize(deckSize:int, withoutCollision:bool):void` que recebe a quantidade de cartas do baralho e se o jogo utilizará somente cartas abertas. Por padrão, o valor do parâmetro `withoutCollision` é falso. Caso seja verdadeiro, o protocolo executará como na proposta de extensão descrita na seção anterior, ou seja, não haverá colisão entre as cartas sendo necessário que o usuário do módulo altere o mapeamento entre os números gerados pelo protocolo e a carta do jogo. Além de determinar o tamanho do baralho que será utilizado durante o jogo e o modo de execução, este método também estabelece a chave pública e a chave privada distribuída aplicando o protocolo de PEDERSEN (1991) descrito na seção 2.3.8 e requerido no passo 2 do protocolo de GOLLE (2005). Por fim, uma chave simétrica, requerida no passo 3 do protocolo de GOLLE (2005), também é estabelecida para cada par de jogadores para o envio de informações seguras. Essas chaves são armazenadas na propriedade `SymmetricKeys:Dictionary<int,BigInteger>`, onde a chave é o identificador do *peer* e o valor é a chave acordada entre os *peers*.

Em seguida, todas as operações do módulo *Mental Poker* podem ser utilizadas. Para isso, a interface `IMentalPoker` fornece uma forma de receber os resultados dessas operações assíncronas. Por exemplo, ao chamar o método `GetOpenCardFromDeck():void` da classe `MentalPoker`, o método `GetOpenCardFromDeck(peerID:int, card:int):void` da interface `IMentalPoker` será chamado passando o valor da carta que foi retirada do baralho e o identificador do *peer* que jogou. A lista a seguir descreve as principais operações que são fornecidas pelo módulo:

- Ler as cartas que o jogador possui na mão: método `GetMyHand():Hand` da classe `MentalPoker`. A classe `Hand` fornece as cartas que um *peer* tem em sua mão. As cartas são representadas por um dicionário, onde a chave é um número que representa a carta criptografada, e o valor é a própria carta;

- Visualizar as cartas criptografadas que os outros jogadores possuem na mão: método `GetPeerHand(peerID:int):Hand` da classe `MentalPoker` que recebe como argumento o identificador do outro *peer*. O jogador somente tem acesso ao valor criptografado da carta, e não ao valor da carta. Isso representa o jogador segurando as cartas de forma que os outros jogadores não conseguem enxergar o valor da carta;

- Visualizar as cartas que estão na mesa: método `GetTableCards():List<int>` da classe `MentalPoker` retorna o valor de cada carta que está na mesa;

- Pegar uma carta aberta do baralho: método `GetOpenCardFromDeck():void`

da classe `MentalPoker`. O método `GetOpenCardFromDeck(peerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. Para o jogador que executou essa ação, o resultado será composto pelo valor da carta e por um identificador vazio, ou seja, é o próprio *peer* que retirou a carta. Para os outros jogadores o resultado será composto pelo identificador do *peer* que retirou a carta e também o valor da carta;

- Pegar uma carta fechada do baralho: método `GetClosedCardFromDeck():void` da classe `MentalPoker`. O método `GetClosedCardFromDeck(peerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. Para o jogador que executou essa ação, o resultado será composto por um identificador vazio e o valor da carta. Para os outros jogadores o resultado será composto pelo identificador do *peer* que retirou a carta com o valor da carta vazio;

- Pegar uma carta aberta do baralho para a mesa: método `GetOpenCardToTable():void` da classe `MentalPoker`. O método `GetOpenCardToTable(peerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. Para o jogador que executou essa ação, o resultado será composto pelo valor da carta e por um identificador vazio, ou seja, é o próprio *peer* que gerou a carta para a mesa. Para os outros jogadores o resultado será composto pelo identificador do *peer* que iniciou a jogada e também o valor da carta;

- Jogar uma carta aberta na mesa: método `PlayOpenCard(card:int):void` da classe `MentalPoker` que recebe como argumento o valor da carta que será jogada na mesa. O método `PlayOpenCard(peerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. O resultado é composto pelo identificador do *peer* que jogou a carta na mesa e também pelo valor da carta. O jogador que realizou a jogada receberá o identificador do *peer* vazio;

- Inserir uma carta fechada da mão no baralho: método `InsertClosedCardInDeck(card:int):void` da classe `MentalPoker` que recebe como argumento o valor da carta que será inserida no baralho. O método `InsertClosedCardInDeck(peerID:int):void` da interface `IMentalPoker` recebe o resultado. O resultado é o identificador do *peer* que inseriu a carta no baralho;

- Pegar uma carta da mão do outro jogador: método `GetClosedCardFromHand(encryptedCard:byte[], peerID:int):void` da classe `MentalPoker` que recebe como argumento o valor da carta

criptografada e o identificador do *peer* que possui essa carta. O método `GetClosedCardFromHand(fromPeerID:int, toPeerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. Para o jogador que recebe a carta, o resultado é composto pelo identificador do *peer* que recebe a carta, nesse caso vazio, o identificador do *peer* que passa a carta e também o valor da carta. Para o jogador que passou a carta, o resultado é composto pelo identificador que recebe a carta, o identificador que passa a carta é vazio, e o valor da carta. Para os outros jogadores que não participam da operação, o resultado é composto pelos identificadores dos *peers*, mas a carta é vazia;

- Receber uma carta da mão do outro jogador: método `ReceiveClosedCardFromHand(peerID:int):void` da classe `MentalPoker` que recebe como argumento o identificador do *peer* que irá passar uma carta. O método `ReceiveClosedCardFromHand(fromPeerID:int, toPeerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. O resultado é semelhante ao descrito na operação anterior;

- Descartar uma carta aberta no monte de descarte: método `DiscardOpenCard(card:int):void` da classe `MentalPoker` que recebe como argumento a carta para descarte. O método `DiscardOpenCard(peerID:int, card:int):void` da interface `IMentalPoker` recebe o resultado. Para o jogador que descartou a carta, o resultado é composto por um identificador vazio e o valor da carta. Para os outros jogadores o resultado é composto pelo identificador do *peer* que descartou a carta e também pelo valor da carta descartada;

- Descartar uma carta fechada no monte de descarte: método `DiscardClosedCard(card:int):void` da classe `MentalPoker` que recebe como argumento a carta para descarte. O método `DiscardClosedCard(peerID:int):void` da interface `IMentalPoker` recebe como resultado o identificador do *peer* que descartou a carta. Para o jogador que realizou essa operação o valor do identificador do *peer* será vazio.

- Inserir as cartas do monte de descarte de novo no baralho: método `MoveDiscardedCardsToDeck():void` da classe `MentalPoker`. O método `MoveDiscardedCardsToDeck(peerID:int):void` da interface `IMentalPoker` recebe como resultado o identificador do *peer* que iniciou a operação. Para o jogador que realizou essa operação o valor do identificador do *peer* será vazio.

- Remapear o protocolo para a rodada seguinte: método `ResetToAnotherRound():void` da classe `MentalPoker`. Este método

implementa a segunda proposta de alteração do protocolo de cartas, ou seja, as cartas que foram jogadas abertas na mesa ao fim da rodada anterior não precisam ser geradas novamente na rodada seguinte. Este método deve ser usado em jogos onde as cartas dos jogadores são jogadas na mesa ao fim de cada rodada. Ao fim de cada rodada, ao chamar este método, o intervalo de números que o protocolo de cartas pode gerar é alterado, sendo o intervalo entre 0 e (intervalo final atual - número de cartas na mesa). Os elementos dos conjuntos L e T também são removidos. A responsabilidade é do usuário em executar a tarefa de atualizar o mapeamento da tabela de números e cartas do jogo.

A classe `ElGamal`, como o nome sugere, tem a responsabilidade de criptografar e descriptografar uma mensagem utilizando a criptografia assimétrica ElGamal. Por meio dos métodos `Encrypt(data:byte[]):byte[]` e `Decrypt(data:byte[]):byte[]` uma mensagem pode ser criptografada e descriptografada. O método `SetKeys(keys:Dictionary<string,BigInteger>):void` recebe as chaves públicas p, g e y e a chave privada x_i daquele jogador. O método `GenerateLargePrime(length:int):BigInteger` cria um número primo de tamanho `length` *bits*.

A classe `MPHistory` armazena as jogadas que ocorreram durante o jogo. O campo `Operation` descreve uma *string* que identifica o nome do método que correspondente a uma ação, como por exemplo "PlayOpenCard", e os campos `FromPeerID` e `ToPeerID` identificam os *peers* que participaram daquela operação.

Por fim, a classe `Message` representa a mensagem trocada entre os *peers* durante o protocolo e a interface `IProtocolCommitted` permite a troca dessas mensagens entre os *peers* assim como identifica cada *peer* participante do protocolo.

3.3.3 Funcionamento interno

O módulo *Mental Poker* inicia seu funcionamento pela chamada do método `Initialize(deckSize:int, withoutCollision:bool):void`, recebendo como parâmetros o número de cartas do baralho e se o protocolo executará somente com cartas abertas evitando colisão entre as cartas. Juntamente com essas duas configurações, este método inicia internamente mais duas operações: definição da chave pública e da chave privada distribuída utilizada no algoritmo ElGamal por meio do protocolo de PEDERSEN (1991) descrito na seção 2.3.8 e também definição de uma chave simétrica entre cada par de jogador. Estas operações são requeridas nos passos 2 e 3 do protocolo de GOLLE (2005), respectivamente.

Para ilustrar estas operações, a Figura 3.7 mostra as mensagens trocadas entre dois jogadores, onde cada um utiliza o módulo via classe `MentalPoker`. A partir

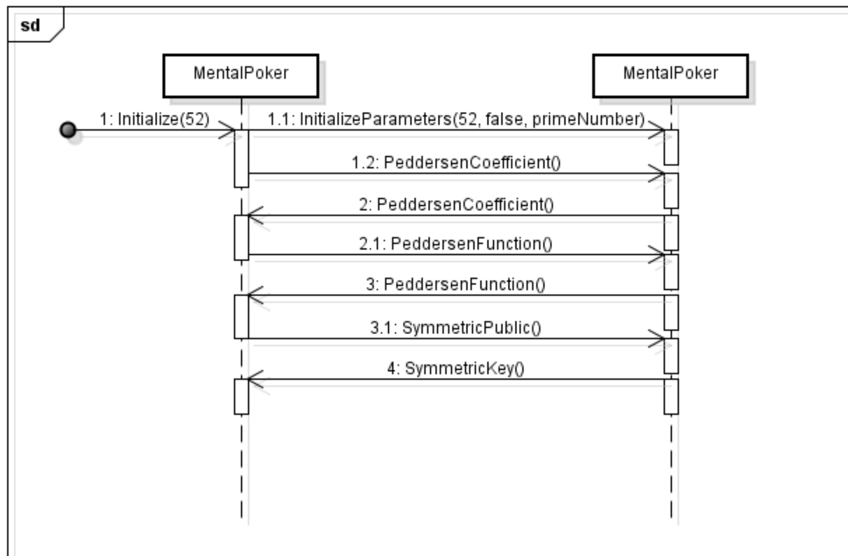

Figura 3.7: Diagrama de sequência de inicialização do módulo *Mental Poker*

do método `Initialize`, uma mensagem `InitializeParameters` é enviada para os outros jogadores informando o número de cartas do baralho, indicação se o protocolo utilizará somente cartas abertas (o que evita colisões) e o número primo que será utilizado na criptografia ElGamal. Em seguida, inicia-se a fase de definição da chave pública e da chave privada distribuída entre os jogadores. As mensagens `PeddersenCoefficient` e `PeddersenFunction` correspondem ao coeficientes do polinômio escolhido pelo jogador e o valor da função, respectivamente. Em seguida, para a definição da chave simétrica utilizada na comunicação entre os dois jogadores no protocolo de cartas, utiliza-se a estratégia de criptografia híbrida (SCHNEIER, 1996, p 86). Em outras palavras, um algoritmo de chave pública é usado como forma de determinar uma chave simétrica entre os participantes. A mensagem `SymmetricPublic` envia a chave pública para o outro jogador, que por sua vez criptografa uma chave simétrica com a chave pública que recebeu. A mensagem `SymmetricKey` envia a chave simétrica criptografada de volta para o outro jogador, que descriptografa por meio da sua chave privada, finalizando o processo de definição da chave simétrica entre eles e também de inicialização do módulo *Mental Poker*.

Depois que o módulo foi inicializado, qualquer operação disponível na classe `MentalPoker` pode ser executada. Por exemplo, a Figura 3.8 apresenta as mensagens que são trocadas internamente no módulo *Mental Poker* entre os jogadores na geração de uma carta aberta. O protocolo é iniciado quando a classe `MentalPoker` recebe uma mensagem `GetOpenCardFromDeck` da classe `Player`. Uma mensagem de notificação chamada `OpenCardSignal` é enviada para o outro jogador indicando que a operação de gerar uma carta aberta deve ser executada. Dessa forma, ambos os jogadores escolhem um número aleatório e o criptografam na classe `ElGamal` com

a mensagem `Encrypt`. Depois disso, ambos os jogadores se comprometem com esse número criptografado aplicando uma função *hash* via mensagem `SHA256` e enviando a mensagem `Commitment`. Em seguida, os jogadores enviam a carta criptografada via mensagem `EncryptedCard` e aplicam o homomorfismo multiplicativo do ElGamal, gerando uma carta que é a soma dos números aleatórios e está na forma de uma tupla ElGamal (a, b). Para determinar se essa carta já saiu, deve-se aplicar o algoritmo de teste de igualdade de texto proposto por JAKOBSSON e SCHNORR (1999). Portanto, quando ambos os jogadores trocam o fator a elevado pela chave privada, ou seja, $a^{chaveprivada}$, é possível verificar se a carta já foi gerada. Em caso positivo, o processo retorna do início. Caso contrário, os jogadores revelam o número escolhido originalmente e também o fator aleatório utilizado na criptografia do ElGamal via mensagem `OriginalCard`. Por fim, a classe `MentalPoker` envia a mensagem `GetOpenCardFromDeck` para a classe `Player` passando o identificador do jogador e também o valor da carta gerada.

Análise de complexidade

Como forma de entender como cada operação escala, a seguir é realizada a análise de complexidade de cada operação do módulo *Mental Poker* com o objetivo de descrever o comportamento dessas operações para um volume grande de jogadores. Assumindo que a variável n denota o número de jogadores do protocolo, as operações podem ser avaliadas da seguinte forma:

1. **Geração distribuída de chaves:** Seguindo o protocolo de PEDERSEN (1991) descrito na seção 2.3.8, cada jogador deve enviar para os outros jogadores 2 mensagens, uma contendo os coeficientes do polinômio e outra contendo o resultado desse polinômio para um número qualquer. Logo, o número total de mensagens é 2 * n * (n - 1), ou seja, n jogadores enviam 2 mensagens para os (n - 1) jogadores restantes. De outra forma, $2n^2$ - $2n$. A complexidade é $O(n^2)$;

2. **Teste de igualdade de texto:** Seguindo o protocolo de JAKOBSSON e SCHNORR (1999), cada jogador envia uma mensagem para os outros jogadores contendo o fator a elevado pela sua chave privada. O fator a pertence à tupla ElGamal (a,b) correspondente à carta criptografada. Logo, o número total de mensagens é n * (n - 1), ou seja, n jogadores enviando uma mensagem para os outros (n - 1) jogadores. Com isso, o número de mensagens é n^2 - n. No protocolo de cartas proposto por GOLLE (2005), o conjunto S mantém n mensagens criptografadas. Para verificar se uma carta já foi gerada no jogo, o teste de igualdade de texto deve ser realizado para cada elemento do con-

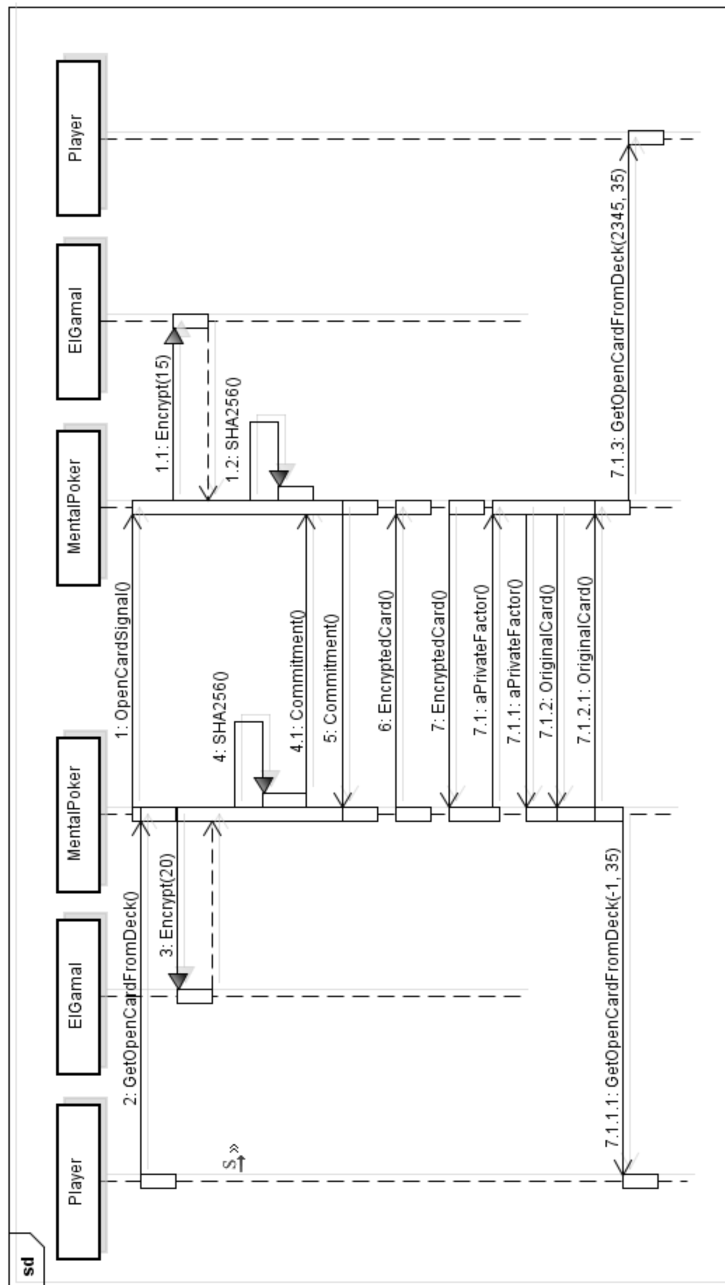

Figura 3.8: Diagrama de sequência de mensagens internas do módulo *Mental Poker* na geração de uma carta aberta

junto S. Portanto, o número total de mensagens é n * (n^2 - n) = $n^3 - n^2$. A complexidade é $O(n^3)$;

3. **Gerar carta fechada:** Para um jogador enviar uma mensagem para todos os outros jogadores, o número de mensagens é (n - 1). Como cada jogador deve enviar duas mensagens que correspondem ao *hash* de comprometimento da carta criptografada e o valor da carta criptografada, o número de mensagens é 2 * n para cada jogador. Portanto, o número total de mensagens é 2 * n * (n - 1), ou seja, cada jogador envia duas mensagens para os outros jogadores. Para essa nova carta criptografada, deve-se verificar por meio do teste de igualdade de texto se a mesma já foi gerada. Logo, deve-se acrescentar $n^3 - n^2$ mensagens de teste de igualdade de texto. Por fim, depois que foi garantido que a carta ainda não foi gerada, cada jogador envia o número aleatório de carta escolhido em segredo para o jogador que está gerando a carta fechada resultando em (n - 1) * 1 mensagens, ou seja, uma mensagem enviada pelo restante dos jogadores ao jogador que está gerando a carta fechada. Portanto, o número total de mensagens é: 2 * n * (n - 1) + $n^3 - n^2$ + (n - 1) * 1 = $n^3 + n^2 - n - 1$. A complexidade é $O(n^3)$;

4. **Gerar carta aberta:** Mesma análise de complexidade realizada acima para gerar uma carta aberta. No entanto, em vez de cada jogador enviar o número aleatório de carta escolhido em segredo para o jogador que está gerando a carta fechada, todos os jogadores trocam entre si esse número aleatório para gerar a carta aberta. Em vez de ser (n - 1) * 1 mensagens, são (n - 1) * n mensagens. Portanto, o número total de mensagens é: 2 * n * (n - 1) + $n^3 - n^2$ + (n - 1) * n = $n^3 + 2n^2 - 3n$. A complexidade é $O(n^3)$;

5. **Jogar carta aberta na mesa:** O jogador que joga uma carta aberta na mesa deve anunciar essa carta para os outros jogadores, ou seja, uma mensagem para (n - 1) jogadores. Em seguida, soma-se o número de mensagens trocadas no teste de igualdade de texto, ou seja, $n^3 - n^2$. Portanto, o número total para jogar uma carta na mesa é (n - 1) + ($n^3 - n^2$) = $n^3 - n^2$ + n - 1. A complexidade é $O(n^3)$;

6. **Inserir carta no baralho:** Para inserir uma carta no baralho, o jogador envia para os outros (n - 1) jogadores a carta que deve ser inserida de volta ao baralho. Portanto, o número total de mensagens é n - 1. A complexidade é $O(n)$;

7. **Pegar ou receber carta da mão do outro jogador:** O jogador que deseja pegar uma carta da mão de outro jogador, deve avisar a todos os jogadores

qual é essa carta, ou seja, deve enviar uma mensagem para (n - 1) jogadores. O jogador que irá retirar uma carta de sua mão, envia uma mensagem com essa carta criptografada para todos os outros jogadores, ou seja, deve enviar uma mensagem para (n - 1) jogadores. Este mesmo jogador agora envia uma mensagem para o jogador que está pegando a carta com o valor da carta e o fator aleatório utilizado para criptografar a carta, ou seja, envia apenas uma mensagem. Para garantir que esta carta de fato pertence àquele jogador, todos os jogadores executam o teste de igualdade de texto, acrescentando $n^3 - n^2$ mensagens. Portanto, o número total de mensagens é (n - 1) + (n - 1) + 1 + $(n^3 - n^2) = n^3 - n^2 + 2n$ - 1. Este resultado também é válido para a operação onde o jogador recebe a carta da mão do outro jogador. A complexidade é $O(n^3)$;

8. **Descartar carta fechada:** Para inserir descartar uma carta, o jogador envia para os outros (n - 1) jogadores a carta que deve ser descartada. Portanto, o número total de mensagens é n - 1. A complexidade é $O(n)$;

9. **Descartar carta aberta:** O jogador que deseja descartar uma carta aberta deve enviar uma mensagem com a carta criptografada para os outros (n - 1) jogadores. Além disso, também deve enviar uma mensagem para os mesmos (n - 1) jogadores com o valor da carta aberta. Para garantir que de fato esta carta pertence àquele jogador, o teste de igualdade de texto deve ser realizado, acrescentando $n^3 - n^2$ mensagens. Portanto, o número total de mensagens é (n - 1) + (n - 1) + $(n^3 - n^2) = n^3 - n^2 + 2n$ - 2. A complexidade é $O(n^3)$.

10. **Inserir cartas do monte de descarte no baralho:** O jogador que deseja inserir as cartas do monte de descarte de novo no baralho deve enviar uma mensagem os outros (n - 1) jogadores. Portanto, o número total de mensagens é n - 1. A complexidade é $O(n)$.

3.3.4 Como usar o módulo

A seguir é apresentado um código fonte de exemplo do uso do módulo *Mental Poker*. O código está escrito na linguagem de programação C#:

```
1   public class Player : IMentalPoker {
2       private MentalPoker _mentalPoker;
3
4       public Player (MentalPoker mp, int deckSize) {
5           _mentalPoker = mp;
5           _mentalPoker.Initialize(deckSize);
```

```
6           _mentalPoker.SetDelegate(this);
7       }
8
9       public void SendMessage(Message message, int peerID)
10       { /* implementação */ }
11
12      public List<int> GetPeersID()
13       { /* implementação */ }
14
15      public List<MPHistory> GetHistory() {
16           return _mentalpoker.GetHistory();
17      }
18
19      public List<int> GetTableCards() {
20           return _mentalpoker.GetTableCards();
21      }
22
23      public Hand GetMyHand() {
24           return _mentalpoker.GetMyHand();
25      }
26
27      public Hand GetPeerHand(int peerID) {
28           return _mentalpoker.GetPeerHand(peerID);
29      }
30
31      public void GetOpenCardFromDeck() {
32           _mentalpoker.GetOpenCardFromDeck();
33      }
34
35      public void GetOpenCardFromDeck(int peerID, int card)
36       { /* implementação */ }
37
38      public void GetClosedCardFromDeck() {
39           _mentalpoker.GetClosedCardFromDeck();
40      }
41
42      public void GetClosedCardFromDeck(int peerID, int card)
43       { /* implementação */ }
44
```

```
45        public void PlayOpenCard(int card) {
46             _mentalpoker.PlayOpenCard(card);
47        }
48
49        public void PlayOpenCard(int peerID, int card)
50         { /* implementação */ }
51
52        public void InsertClosedCardInDeck(int card) {
53             _mentalpoker.InsertClosedCardInDeck(card);
54        }
55
56        public void InsertClosedCardInDeck(int peerID)
57         { /* implementação */ }
58
59        public void DiscardOpenCard(int card) {
60             _mentalpoker.DiscardOpenCard(card);
61        }
62
63        public void DiscardOpenCard(int peerID, int card)
64         { /* implementação */ }
65
66        public void DiscardClosedCard(int card) {
67             _mentalpoker.DiscardClosedCard(card);
68        }
69
70        public void DiscardClosedCard(int peerID)
71         { /* implementação */ }
72
73        public void GetClosedCardFromHand
74                      (int peerID, byte[] encrytedCard) {
75             _mentalpoker.
76                GetClosedCardFromHand(peerID, encryptedCard);
77        }
78
79        public void GetClosedCardFromHand
80                      (int fromPeerID, int toPeerID, int card)
81         { /* implementação */ }
82
83        public void ReceiveClosedCardFromHand(int peerID) {
```

```
84              _mentalpoker.
85                  ReceiveClosedCardFromHand(peerID);
86          }
87
88          public void ReceiveClosedCardFromHand
89                          (int fromPeerID, int toPeerID, int card)
90           { /* implementação */ }
91      }
```

Para a classe `Player` usar o módulo *Mental Poker* é necessário implementar a interface `IMentalPoker` e fazer agregação com a classe `MentalPoker`. A interface `IProtocolCommitted` é implementada nas linhas 9 e 12, enviando mensagens para outros *peers* e informando o identificador de cada *peer* participante do protocolo.

O construtor, na linha 4, recebe como parâmetros um tipo `MentalPoker` e a quantidade de cartas utilizadas no baralho. Na linha 5 a classe `MentalPoker` é inicializada com o número de cartas do baralho. Como não foi passado o valor do parâmetro `withoutCollision`, o valor padrão é falso, ou seja, o protocolo pode utilizar cartas fechadas e com colisão. Na linha 6 é declarado a delegação, onde a classe do tipo `MentalPoker` irá delegar o envio de mensagens pela classe `Player`.

Todos os outros métodos representam alguma operação do módulo definida na própria classe `MentalPoker`. Como exemplo, na linha 32 o método `GetOpenCardFromDeck` é invocado e, por se tratar de uma operação assíncrona como a maioria das operações do módulo, o resultado é obtido na linha 35 com a chamada do método `GetOpenCardFromDeck(int peerID, int card):void` da interface `IMentalPoker`, recebendo como argumentos o identificador do *peer* que realizou a jogada e também a carta retirada. As outras operações seguem o mesmo padrão e são descritas em detalhes na seção 3.3.2 que apresenta a modelagem do módulo.

Para ilustrar algumas interações, a Figura 3.9 apresenta algumas operações com o módulo *Mental Poker* entre dois jogadores. Primeiro um dos jogadores pede uma carta aberta do baralho via chamada `GetOpenCardFromDeck`. A classe `MentalPoker` envia para os dois jogadores o resultado dessa operação, passando o identificador do jogador que pediu a carta, nesse caso identificador de número 2345, e o valor da carta, 23. O próprio jogador que iniciou a jogada recebe -1 como identificador do jogador, pois foi ele mesmo que jogou. O jogador continua pedindo uma carta fechada do baralho via chamada `GetClosedCardFromDeck`. A classe `MentalPoker` envia para os dois jogadores o resultado dessa operação, porém omite o valor da carta para o outro jogador, pois a carta retirada do baralho estava fechada. Dessa forma, o jogador que retirou a carta recebe o valor 14 e o outro jogador recebe o

valor -1. Por último, o jogador joga uma carta na mesa via chamada `PlayOpenCard`, e ambos os jogadores são notificados dessa operação pela classe `MentalPoker`.

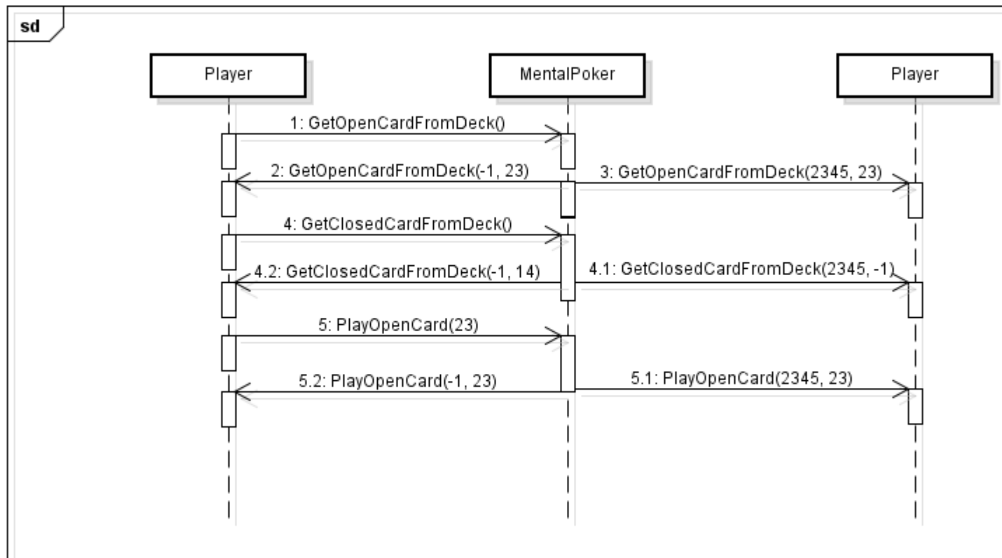

Figura 3.9: Diagrama de sequência de exemplo de uso do módulo *Mental Poker*

3.4 Decisões de *desing* e padrões adotados

A modelagem dos módulos exigiu que algumas decisões arquiteturais fossem tomadas a fim de deixar o módulo simples de usar, fácil de entender e manter do código. O requisito fundamental foi deixar simples o uso dos módulos da biblioteca. Para usar qualquer módulo, basta apenas implementar uma interface, `IDiceRolling` para o módulo *Dice Rolling* ou `IMentalPoker` para o módulo *Mental Poker*. Todas essas interfaces herdam da interface base `IProtocolCommitted` que tem dois métodos abstratos, um responsável por enviar uma mensagem para outro *peer* e o outro que retorna o identificador de cada *peer* participante do protocolo. Portanto, a estrutura de herança entre as interfaces deixa claro que as funcionalidades de cada módulo foram construídas a partir de uma interface base que suporta a comunicação entre os jogadores.

Para o uso da biblioteca, preferiu-se a implementação de uma interface em vez de herdar de uma outra classe. Supondo que para usar um módulo o usuário tivesse que herdar de uma superclasse desse módulo, provavelmente a subclasse estaria impedida de assumir outros comportamentos via herança, já que é muito provável que a linguagem orientada a objetos usada não suportaria herança múltipla [6]. A

[6]Herança múltipla é uma propriedade das linguagens orientadas a objeto que permitem que uma subclasse herde características e atributos de mais de uma superclasse. Linguagens populares que possuem herança múltipla são C++, Perl e Python. Muitos questionam as desvantagens da herança múltipla, como aumento da complexidade e ambiguidade caso um mesmo método seja

vantagem de usar uma interface é permitir que a classe usuária fique livre para implementar outras interfaces e também herdar de outra superclasse.

Além de evitar a herança múltipla, a modelagem seguiu em particular o segundo princípio de *design* orientado a objetos proposto por GAMMA *et al.* (1995) como técnica para reutilização de funcionalidades e comportamentos: prefira composição em vez de herança (GAMMA *et al.*, 1995, p 32). A herança apresenta desvantagens como não poder mudar, em tempo de execução, a implementação da superclasse. A subclasse fica dependente da implementação da superclasse que, ao mudar sua implementação, irá forçar que a subclasse mude também. Por outro lado, a composição permite a mudança de implementação em tempo de execução por meio da referência a outro objeto que implementa uma determinada interface de contrato. A partir disso, outra técnica de implementação foi utilizada em conjunto com a implementação das interfaces `IDiceRolling` ou `IMentalPoker`: delegação [7]. Delegação é uma composição de objetos que funciona como um mecanismo de reutilização de código, onde um objeto delega uma determinada tarefa para outro objeto (GAMMA *et al.*, 1995, p 32). Por exemplo, em vez de uma classe `Window` herdar de uma classe `Rectangle`, a classe `Window` pode reutilizar o comportamento da classe `Rectangle` mantendo uma referência para essa classe e delegando tarefas específicas dessa classe (GAMMA *et al.*, 1995, p 33). Por exemplo, no contexto do módulo *Dice Rolling*, a classe `DiceRolling` delega para a classe usuária o envio de uma mensagem para outro *peer* via método `SendMessage(message: Message, peerID: int): void` da interface `IProtocolCommitted`, pois não cabe a classe `DiceRolling` saber enviar mensagem, mas sim implementar o protocolo criptográfico. A classe usuária é que tem a responsabilidade de saber enviar uma mensagem para outro *peer*, independente do meio de comunicação utilizado.

No momento em que o usuário envia uma notação de dado para o módulo *Dice Rolling* a fim de obter o resultado dos dados, a classe `DiceRolling` recebe essa notação de dados, executa e delega determinadas tarefas. Essa sequência de tarefas é descrita pela aplicação do padrão de projeto *Template Method*. O padrão de projeto *Template Method* define o esqueleto de um algoritmo, delegando alguns passos desse algoritmo para subclasses ou componentes agregados que definem certos passos desse algoritmo sem alterar a estrutura desse algoritmo (GAMMA *et al.*, 1995, p 360). Nesse contexto, a classe `DiceRolling` recebe uma notação de dado, delega para outro componente a validação dessa notação e delega o cálculo do número de dados que essa notação exige. A partir dessa quantidade de dados, delega para a subclasse o cálculo dos números seguros via protocolo criptográfico. Em seguida, delega o cálculo do resultado com base na notação de dado e nos números seguros. Pode-

implementado por mais de uma superclasse.

[7] Tradução para o termo em inglês *delegation*

se perceber que o padrão de projeto *Template Method* estabelece uma estrutura de controle invertido que é referenciada como "Princípio Hollywood"[8], que significa que a superclasse invoca operações nas subclasses ou em outros componentes agregados, nunca ao contrário (GAMMA *et al.*, 1995, 362)

Por fim, o usuário tem a possibilidade de estender certas funcionalidades do módulo *Dice Rolling* por meio da implementação da sua própria lógica, substituindo a funcionalidade original do módulo [9]. O protocolo criptográfico que gera os dados pode ser estendido. Por padrão, o módulo *Dice Rolling* implementa uma versão do protocolo criptográfico utilizando função de mão única. No entanto, o usuário desse módulo pode implementar sua própria versão do protocolo criptográfico, por exemplo implementando uma versão que utilize criptografia simétrica como forma de garantir o *bit commitment*. Para isso, a classe que implementa esse novo protocolo deve herdar da classe `DiceRolling` e implementar os seus métodos abstratos `CalculateSecureNumbers(numberOfDices: int): void` e `ReceiveMessage(message: Message, peerID: int): void`. O primeiro método tem a responsabilidade de gerar, por meio do protocolo criptográfico, `numberOfDices` números que serão usados para o calculo da notação de dado. O segundo método tem a responsabilidade de capturar uma mensagem enviada por outro *peer* a fim de tratar essa mensagem a partir do passo atual do protocolo.

Outra forma de estender o módulo *Dice Rolling* é definir a sua própria notação de dado. Por padrão, o módulo implementa uma versão da notação padrão de dados descrita na seção 3.2.2. Para o usuário definir sua própria notação de dado, deve-se implementar a interface `IDiceNotationParser` juntamente com seus métodos abstratos. A responsabilidade dessa interface é definir um contrato onde seja possível identificar se a notação de dado é válida ou não, determinar quantos dados aquela notação de dado requer, e a partir da notação de dado e um conjunto de números gerados via protocolo criptográfico, calcular os resultados dessa jogada. Depois de criada uma classe que implemente esse contrato, basta configurar essa notação como a notação padrão do módulo *Dice Rolling* chamando o método `SetDiceNotationParser(DiceNotationParser: IDiceNotationParser): void`, passando como argumento a nova classe criada para sobrescrever a notação de dado original.

[8]Tradução para o termo em inglês *The Hollywood Principle*

[9]Seguindo o princípio Aberto/Fechado, traduzido do termo em inglês *Open/Closed principle*, as classes estão abertas para extensão, mas fechadas para modificação (MEYER, 1988). Ou seja, em vez de alterar diretamente o código fonte, a funcionalidade deve ser adicionada pela implementação de uma nova classe que é adicionada por meio do polimorfismo criado via interface ou herança.

Capítulo 4

Implementação

Este capítulo descreve como foi implementada a biblioteca proposta no capítulo anterior. Para tal, foi utilizado como Ambiente de Desenvolvimento Integrado a ferramenta Microsoft Visual Studio Ultimate 2012[1], utilizando C# 5.0[2] como linguagem de programção em conjunto com a plataforma de desenvolvimento e execução de aplicações Microsoft .NET Framework[3], versão 4.5. A biblioteca implementada é composta por 15 classes, 4 interfaces, 2 *enumerators* e com aproximadamente 3.100 linhas.

4.1 Estrutura do *peer*

Com o propósito de usar a biblioteca proposta no capítulo anterior, foi necessário a implementação de um *peer* que pudesse se comunicar com outro *peer*. Em cima dessa arquitetura de envio e recebimento de mensagens na forma de *array* de *bytes*, foi possível implementar os módulos da biblioteca e também os protocolos criptográficos. A Figura 4.1 apresenta o diagrama de classes que compõem da estrutura de um *peer*:

A classe `Peer` é composta por duas agregações, uma com a classe `Client` e uma com a classe `Server`. A classe `Peer` contém uma lista de objetos do tipo `Client`, onde cada instância representa um canal de comunicação com um determinado *peer*. A classe `Server` é designada a receber e a tratar conexões de novos *peers*.

Assim que a classe `Peer` é instanciada, uma outra *thread* é criada para executar uma instância da classe `Server`, que fica em *loop* infinito esperando um novo *peer* conectar. Dessa forma, a classe `Peer` pode tratar de outros eventos, como *click* de mouse, entrada de texto, tratamento de eventos, etc, sem que a classe `Server` bloqueie a execução da *thread* principal. Assim que um novo *peer* se conecta, a classe `Server` cria uma nova *thread* que trata exclusivamente do recebimento de

[1] http://www.microsoft.com/visualstudio/eng/products/visual-studio-ultimate-2012
[2] http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx
[3] http://www.microsoft.com/net

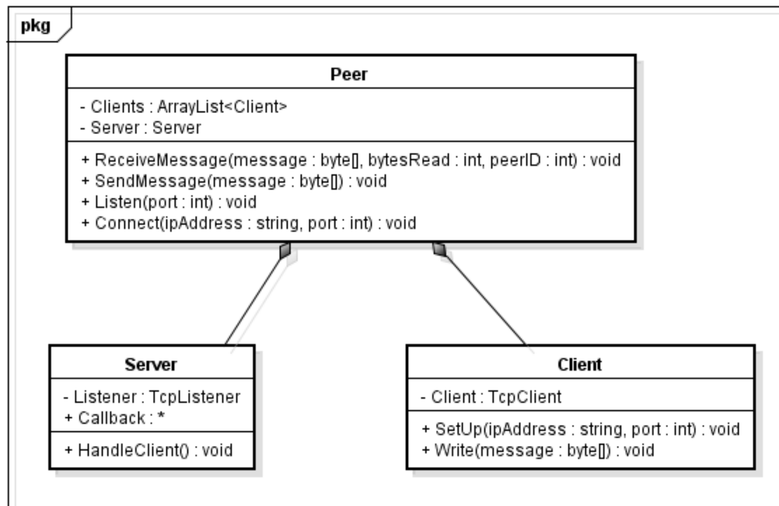

Figura 4.1: Diagrama de classes do *peer*

mensagens desse novo *peer*. Portanto, a classe `Peer` executa na *thread* principal, a classe `Server` executa em uma outra *thread* em *loop* infinito esperando por conexões de novos *peers*, e cada conexão de um novo *peer* executa em uma outra *thread*. Por exemplo, caso um *peer* esteja conectado a três outros *peers*, há uma *thread* principal executando a instância de *peer*, uma outra *thread* executando a instância de server e outras três *threads* que tem a responsabilidade de receber as mensagens de cada *peer* conectado.

A Figura 4.2 mostra a ordem temporal das mensagens no processo de criação e troca de mensagens de um *peer*:

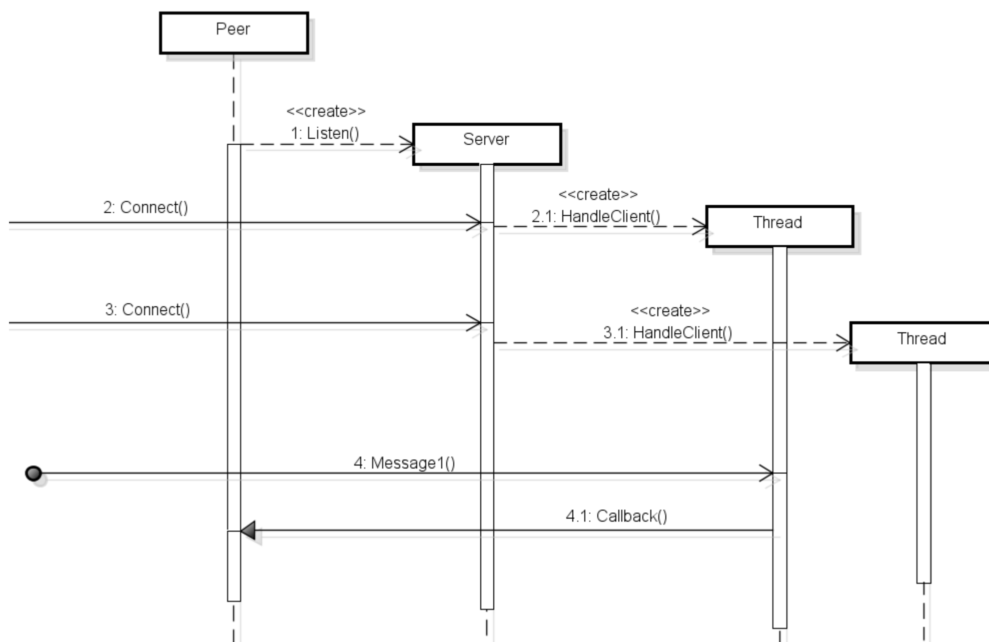

Figura 4.2: Diagrama de sequência de conexão e recebimento de mensagem de um *peer*

Ao instanciar a classe `Peer`, por meio da mensagem `Listen`, uma nova instância da classe `Server` é instanciada para escutar, em uma determinada porta, por novos *peers* que desejam se conectar. Quando um novo *peer* se conecta por meio da mensagem `Connect`, a instância da classe `Server` cria uma nova *thread* por meio da mensagem `HandleClient` para escutar pelas mensagens desse novo *peer*. Esse processo se repete para cada novo *peer* que deseja se conectar. Quando uma mensagem é recebida, a *thread* secundária envia essa mensagem para a *thread* principal via mensagem `Callback`. Dessa forma, a *thread* principal pode tratar a mensagem recebida do outro *peer* de acordo com o contexto da aplicação.

Para implementar a estrutura de um *peer*, a linguagem de programação C# provê os *namespaces* System.Net[4] e System.Net.Sockets[5] que são ideais para programação em rede, encapsulando e gerenciando a programação via *socket*[6], além de encapsular os detalhes de conexão TCP[7] na internet.

A classe `Client` utiliza como base a classe `TcpClient`[8] que provê métodos para conectar, enviar e receber um fluxo de dados pela rede. A classe `TcpClient` é utilizada em conjunto com a classe `IPEndPoint`[9] para estabelecer um ponto-final na rede por meio do endereço IP e um número de porta. Dessa forma, a classe `Client` tem a capacidade de se conectar e enviar mensagens na rede.

A classe `Server` utiliza como base a classe `TcpListener`[10] que provê métodos para escutar e aceitar requisições de conexão na rede. A classe `IPEndpoint` também é utilizada em conjunto com a classe `TcpListener` para estabelecer um ponto-final na rede. Com isso, a classe `Server` tem a capacidade de receber mensagens enviadas pela rede.

4.2 Módulo *Dice Rolling*

No módulo *Dice Rolling*, a classe `DROneWayFunction` é que de fato implementa o protocolo criptográfico utilizando função de mão única como estratégia de *bit commitment*. Essa classe não guarda em que passo a execução do protocolo se encontra, pois essa informação é enviada junto com cada mensagem do tipo `Message` que o *peer* recebe. Portanto, a classe `DROneWayFunction` tem a seguinte estrutura:

[4] http://msdn.microsoft.com/en-us/library/system.net.aspx

[5] http://msdn.microsoft.com/en-us/library/system.net.sockets.aspx

[6] Soquete de rede, tradução do termo em inglês *socket*, é um ponto-final do fluxo de comunicação entre processos através de uma rede de computadores.

[7] O Protocolo de Controle de Transmissão, tradução do termo em inglês *Transmission Control Protocol* - TCP, é um protocolo de rede que verifica se os dados em uma comunicação são enviados de uma forma correta, na mesma sequência e sem erros.

[8] http://msdn.microsoft.com/en-us/library/1612451t.aspx

[9] http://msdn.microsoft.com/en-us/library/fzszfbba.aspx

[10] http://msdn.microsoft.com/en-us/library/zsyxy9k2.aspx

```
1   public class DROneWayFunction : DiceRolling {
2     public override void CalculateSecureNumbers
3                                      (int NumberOfDice) {
4       /* gera números aleatórios */
5       /* gera os números hash de comprometimento */
6       var message =
7        new Message(MessageMethod.FirstDiceHashCommitment, hash);
8       Delegate.SendMessage(message);
9     }
10
11    public override void ReceiveMessage
12                        (Message message, int peerID){
13      switch (message.Method) {
14        case MessageMethod.FirstDiceHashCommitment: {
15          /* guarda os numeros hash de comprometimento
16            do peer que iniciou a jogada */
17
18          /* gera numeros aleatorios e a partir desses
19            gera os números hash de comprometimento */
20   var message =
21            new DRMessage(MessageMethod.LastDiceHashCommitment,
22                                                          hash);
23          Delegate.SendMessage(message);
24        }
25        case MessageMethod.LastDiceHashCommitment: {
26          /* guarda os números hash de comprometimento
27          dos outros peers */
28
29          var message =
30            new DRMessage(MessageMethod.FirstDiceHashReveal,
31                                             originalValues);
32         }
33         case MessageMethod.FirstDiceHashReveal: {
34           /* recebe os valores aleatorios originais do peer
35            que iniciou a jogada e verifica se os números hash
36            comprometidos são iguais aos valores
37             hash dos números originais */
38
39            /* envia os números gerados para
```

```
40                a super classe DiceRolling calcular
41                os resultados */
42
43                var message =
44                 new DRMessage(MessageMethod.LastDiceHashReveal,
45                                   originalValues);
46            }
47            case MessageMethod.LastDiceHashReveal: {
48              /* recebe os valores originais dos outros peers
49              e verifica se os números hash comprometidos
50              são iguais aos valores hash dos números
51              originais */
52
53              /* envia os números gerados para
54              a super classe DiceRolling calcular
55              os resultados */
56            }
57          }
58        }
59    }
```

Suponha a participação de dois jogadores no protocolo, Alice e Bob. Supondo que Alice seja o jogador que inicia a jogada, Alice gera os números aleatórios e em seguida se compromete com esses números por meio de uma função de mão única, gerando os valores *hash* que são enviados para o jogador Bob na linha 8. Quando Bob recebe a mensagem, verifica que o método da mensagem indica que Alice está iniciando uma jogada, o que indica que Bob deve guardar os valores enviados por Alice, gerar seus números aleatórios e também seus respectivos valores *hash*. Bob se compromete com seus valores *hash* enviando uma mensagem para Alice na linha 23. Alice recebe a mensagem, guarda os valores *hash* enviados por Bob e envia uma outra mensagem para Bob na linha 30. Essa mensagem contém os números aleatórios gerados no primeiro passo. Bob recebe a mensagem que indica e verifica se o valores *hash* dos números aleatórios enviados por Alice são iguais aos valores *hash* enviados quando Alice iniciou a jogada. Em seguida, Bob envia uma mensagem para Alice enviando os números aleatórios gerados por Bob. Alice recebe essa mensagem e verifica se os valores *hash* dos números aleatórios enviados por Bob são iguais aos valores *hash* enviados por Bob. Tanto Alice quanto Bob enviam os números aleatórios gerados pelo protocolo para a super classe `DiceRolling` que, a partir desses números e da notação de dados, calcula o resultado da jogada.

Para implementar o protocolo foram utilizadas algumas classes da plataforma .Net. A classe `SHA256`[11] é utilizada como função de mão única. O valor *hash* é um valor único de tamanho fixo representando um número aleatório, onde dois *hashes* são iguais somente se os números aleatórios forem iguais. O valor *hash* gerado pelo algoritmo é representado por 256 *bits*.

Para gerar os números aleatórios foi utilizada a classe `Random`[12]. Essa classe gera números pseudo aleatórios, ou seja, gera uma sequência de números que atende a certos requisitos estatísticos de aleatoriedade.

Por fim, para que uma mensagem seja enviada, é necessário converter os números aleatórios e seus respectivos valores *hash* para um *array* de *bytes*. Com esse objetivo foi utilizada a classe `BitConverter`[13] que converte tipos primitivos da linguagem C# num *array* de *bytes* e vice versa.

4.3 Módulo *Mental Poker*

De fato a implementação deste módulo é a primeira implementação conhecida do protocolo proposto por GOLLE (2005)[14]. No módulo *Mental Poker*, toda implementação está baseada na criptografia assimétrica ElGamal que utiliza um número primo p. Portanto, o primeiro requisito a ser satisfeito é a criação de um número primo p de forma que seja possível utilizar o protocolo de cartas. Ao inicializar o módulo, um número primo p deve ser gerado. A ideia é escolher aleatoriamente um número grande n e testar se este número é primo. Para realizar este teste, este trabalho utiliza o teste de primalidade Miller-Rabin RABIN (1980), cujo algoritmo produz um teste probabilístico. O algoritmo proposto por RABIN (1980) é uma modificação do algoritmo de MILLER (1976). O teste de primalidade de um número n é realizado da seguinte forma:

1. Sendo n - 1 um número par, o mesmo pode ser escrito como $2^s d$, onde s e d são número inteiros positivos e d é um número ímpar. Seja também um número aleatório $a < n$;

2. Testar se $a^d \equiv 1 (mod\ n)$ e;

3. Testar se $a^{2^r d} \equiv -1 (mod\ n)$ para $0 \leq r \leq s$ - 1.

Se os dois testes falharem, estão o número n é composto. Se os resultados dos testes forem positivos, então há uma grande probabilidade de o número n ser

[11] http://msdn.microsoft.com/en-us/library/9x979460.aspx
[12] http://msdn.microsoft.com/en-us/library/ts6se2ek(v=vs.102).aspx
[13] http://msdn.microsoft.com/en-us/library/3kftcaf9.aspx
[14] Comunicação pessoal, via correio eletrônico no dia 20 de Agosto de 2013, com o pesquisador e autor do protocolo de cartas Philippe Golle

primo. Para aumentar esta probabilidade, o teste deve ser realizado diversas vezes para diferentes valores de a. Como apresentado na seção 3.3.2 de Modelagem, a classe `ElGamal` possui o método `GenerateLargePrime(length:int):BigInteger` que gera um número primo de tamanho `length` *bits* por meio do algoritmo acima. Como ideia do tamanho necessário para o número primo p utilizado no protocolo de cartas via criptografia ElGamal, assuma 5 jogadores e um baralho de 52 cartas. Assuma que o elemento gerador g seja 2. Dessa forma, em uma rodada onde uma carta é gerada, se cada jogador escolher o maior número, 51, o homomorfismo multiplicativo resultará em $2^{51} * 2^{51} * 2^{51} * 2^{51} * 2^{51} = 2^{255}$. Portanto, o número primo deve ser um número com o tamanho de 255 *bits*. O número abaixo é um exemplo de um número primo com 255 *bits* de tamanho:

$$926336713898529563388567880069503262826159877325124512315660672063305037119923$$

A plataforma .Net não tem nenhuma implementação do algoritmo assimétrico ElGamal, portanto foi necessário implementar esse algoritmo. De fato, a classe `BigInteger`[15] foi fundamental para a implementação do algoritmo. Com esta classe foi possível criar e manipular grandes números via métodos da própria classe, com destaque para o método `ModPow(value:BigInteger,exponent:BigInteger,modulus:BigInteger)` que executa a expressão $value^{exponent} mod\ modulus$. A única implementação encontrada do algoritmo ElGamal estava no livro *Programming .Net Security* do conhecido autor Adam Freeman (FREEMAN e JONES, 2003). A implementação da classe `ElGamal` se inspirou naquela implementação, com o foco em deixar o código mais enxuto e também substituir uma antiga classe `BigInteger`[16] escrita por Chew Keong TAN pela classe `BigInteger` implementada anos depois na plataforma .Net.

No protocolo de cartas proposto por GOLLE (2005) descrito na seção 2.3.10, o passo 2 define que cada par de jogadores (P_i, P_j) deve acordar uma chave secreta $k_{i,j}$ para ser usada em uma criptografia simétrica, o que torna possível uma comunicação segura entre os jogadores. Para implementar este passo, foi utilizado a estratégia de criptografia simétrica, ou seja, utiliza-se um algoritmo assimétrico para trafegar a chave de um algoritmo simétrico (SCHNEIER, 1996, p 86). O algoritmo ElGamal implementado na classe `ElGamal` serviu como o algoritmo assimétrico. A classe `DESCryptoServiceProvider`[17] da plataforma .Net foi utilizada como o algoritmo simétrico, especificamente o algoritmo DES.

[15] http://msdn.microsoft.com/en-us/library/system.numerics.biginteger.aspx
[16] http://www.codeproject.com/Articles/2728/C-BigInteger-Class
[17] http://msdn.microsoft.com/en-us/library/system.security.cryptography .descryptoserviceprovider.aspx

Colocar em prática a teoria geralmente expõe detalhes que não são conhecidos ou percebidos de início. No passo 9 do protocolo de cartas proposto por GOLLE (2005) descrito na seção 2.3.10, verifica-se se duas cartas são iguais por meio do protocolo de teste de igualdade de texto proposto por JAKOBSSON e SCHNORR (1999). Assumindo que o conjunto L contenha a carta $E(c_1) = (g^{r_1}, g^{c_1}y^{r_1}) = (a_1, b_1)$ e que a carta $E(c_2) = (g^{r_2}, g^{c_2}y^{r_2}) = (a_2, b_2)$ foi gerada e deve ser testada, aplica-se o homomorfismo multiplicativo do ElGamal $\frac{E(c_1)}{E(c_2)} = (g^{r_1-r_2}, g^{c_1-c_2}y^{r_1-r_2}) = (a,b)$. O primeiro cuidado deve ser garantir que r_1 seja maior que r_2, caso contrário o resultado será um número não inteiro, pois $g^{r_1-r_2} = g^{-r} = \frac{1}{g^r}$, sendo $r = r_1 - r_2$. Como forma de garantir este requisito, a ideia foi comparar os fatores a das duas cartas criptografadas, ou seja, se $a_2 > a_1$. Como o número gerador g é igual, é possível verificar qual fator aleatório r_i é maior sem conhecer o seu valor. Se r_2 for maior que r_1, a solução é incrementar o valor de r_1 com algum fator conhecido de forma que não modifique a mensagem original criptografada. Para isso, o número 1 é criptografado com um fator aleatório r conhecido, como por exemplo, o valor 5. Criptografar a mensagem 1 com o fator aleatório 5 resulta em $(g^5, 1 * y^5) = (g^5, y^5)$. Utilizando o homomorfismo multiplicativo, altera-se a mensagem $E(c_1)$ como $E(c_1)*E(1) = (g^{r_1}, g^{c_1}y^{r_1})*(g^5, y^5) = (g^{r_1+5}, g^{c_1}y^{r_1+5})$. Novamente os fatores a_i são comparados, e este procedimento repete-se enquanto $a_2 > a_1$.

Além de garantir que $a_2 < a_1$, outra situação deve ser considerada. Se a carta $c_2 > c_1$, novamente o resultado será um número não inteiro, pois $g^{c_1-c_2} = g^{-c} = \frac{1}{g^c}$. Portanto, é necessário testar o homomorfismo multiplicativo nas duas combinações, ou seja, testar $\frac{E(c_1)}{E(c_2)}$ e também $\frac{E(c_2)}{E(c_1)}$, sempre garantindo que a subtração dos números aleatórios resultem num número positivo como descrito acima.

Assim como apresentado na implementação do módulo *Dice Rolling*, a classe `SHA256`[18] é utilizada como função de mão única na fase de comprometimento com a carta criptografada no passo 6 do protocolo de GOLLE (2005). O valor *hash* gerado pelo algoritmo é representado por 256 *bits*. Na geração de números aleatórios também é utilizada a classe `Random`[19]. No envio de mensagens, a classe `BitConverter`[20] é utilizada para converter tipos primitivos da linguagem C# num *array* de *bytes* e vice versa.

[18]http://msdn.microsoft.com/en-us/library/9x979460.aspx
[19]http://msdn.microsoft.com/en-us/library/ts6se2ek(v=vs.102).aspx
[20]http://msdn.microsoft.com/en-us/library/3kftcaf9.aspx

Capítulo 5

Experimento

Neste capítulo é apresentado um estudo experimental conduzido para avaliar o comportamento e a eficiência da biblioteca desenvolvida neste trabalho, assim como avaliar os protocolos criptográficos propostos. Dessa forma, tanto para o módulo *Dice Rolling* quanto para o módulo *Mental Poker*, serão descritos os objetivos do experimento, o cenário em que este experimento foi desenvolvido e, por fim, a apresentação dos resultados obtidos e uma análise geral sobre os mesmos.

5.1 Configuração do ambiente

Para executar, testar e avaliar os resultados sobre o uso da biblioteca proposta nesse trabalho, o experimento descrito a seguir foi realizado num PC com sistema operacional Windows 7, com processador Intel Core i5 2.6GHz e memória RAM de 4 GB.

5.2 Experimento no módulo *Dice Rolling*

O objetivo é testar a implementação do módulo *Dice Rolling*, executar determinados cenários de uso por meio de jogadas com variação na notação de dados e também com variação na quantidade de jogadores. Serão realizadas consultas no histórico de jogadas para saber, por exemplo, qual jogador jogou os dados na terceira jogada, qual foi o resultado fechado da segunda jogada, qual o comentário e o resultado aberto da última jogada, etc. Além disso, a partir de jogadas retiradas de jogos reais, o experimento tem o objetivo de avaliar certas métricas, como o tempo médio de execução de uma jogada no protocolo, o tráfego médio em *bytes* para executar uma jogada, o tempo médio necessário para realizar as operações de criptografia e também a quantidade de mensagens trocadas entre os jogadores. A seguir é apresentado o cenário do experimento e os resultados obtidos.

5.2.1 Cenário do experimento

O experimento foi composto por *peers*, onde cada *peer* corresponde a uma aplicação console que está executando localmente e escutando em uma determinada porta. Por exemplo, três *peers*, Alice, Bob e Carol escutam em localhost:8080, localhost:8081, localhost:8082 respectivamente. O relacionamento entre os *peers* é um grafo completo, ou seja, cada *peer* está conectado com todos os outros *peers* do sistema.

Ao executar a aplicação console que corresponde a um *peer*, foi feita a conexão entre os *peers*. Depois que todos os *peers* se conectaram uns com os outros, foi possível a entrada de comandos na aplicação console. O comando de jogar os dados tem a forma <notação de dado, comentário>, onde o campo comentário era opcional. Outra opção era executar comandos de consulta ao histórico das jogadas que foram executadas nas rodadas passadas da seguinte forma:

- `list`: retorna a lista de todas as rodadas do histórico. Cada rodada é composta pelo nome do jogador que iniciou a jogada, a notação de dado, o resultado fechado, o resultado aberto e o comentário da jogada;
- `list/rodada`: retorna o histórico completo de uma determinada rodada. Por exemplo, o comando `list/3` retorna o histórico da terceira rodada;
- `list/rodada/atributo`: retorna um valor específico de uma determinada rodada. Por exemplo, o comando `list/3/resultadoAberto` retorna o resultado aberto da terceira rodada.

O primeiro passo foi executar cada notação de dado da Tabela 5.1, tanto no modo normal quanto no modo em lote. Essas notações de dados foram escolhidas pois são muito comuns nos mais diversos jogos. O objetivo é averiguar os resultados gerados por cada jogada, como o resultado aberto, o resultado fechado, o comentário e o jogador que realizou a jogada.

Tabela 5.1: Notações de dado comuns

Identificador	Notação de dado	Nº de dados	Comentário
ND1	1d6	1	Tirando a sorte
ND2	2d6	2	Matando o monstro
ND3	3d6	3	Atirando no inimigo
ND4	4d6 - L	4	Maior que 10 saio da prisão!
ND5	1d10	1	Jogada bônus!
ND6	2d10	2	Força de defesa
ND7	1d100	1	Causando dano no inimigo
ND8	1d20	1	Se for par saio do jogo

O segundo passo foi executar uma massa maior de jogadas. As jogadas foram retiradas de jogos reais do site *Roleplay online*[1] que atua como uma comunidade de jogos *RPG* que são jogados por meio de *posts* ou mensagens enviadas pelos jogadores. A comunidade já registrou 6.155 jogos e 7.204.159 mensagens. Foram escolhidos três jogos: *Dungeons & Dragons*[2], *Vampire The Masquerade*[3] e *Pathfinder*[4]. Cada jogada foi executada no modo normal e em lote com a participação de 2, 3, 4 e 5 *peers*. A Tabela 5.2 mostra, para cada jogo, o total de jogadas e também o total de dados jogados. O sistema de dados predominante para os jogos *Dungeons & Dragons* e *Pathfinder* é d20, e para o jogo *Vampire The Masquarade* é d10.

Tabela 5.2: Jogos reais utilizados no experimento

Jogo	Total de jogadas	Total de dados
Dungeons & Dragons	651	1284
Pathfinder	688	930
Vampire The Masquerade	278	1345

5.2.2 Resultados

A Tabela 5.3 apresenta os resultados das jogadas para o modo normal e para o modo em lote, exibindo o resultado aberto, ou seja, o valor individual de cada dado, e também o resultado final que é a avaliação da notação de dado com o valor de cada dado:

A partir disso foi possível exercitar a capacidade do módulo *Dice Rolling* de armazenar o histórico das jogadas. Por exemplo, para saber o comentário feito na quarta jogada, a consulta `list/4/comentario` retorna o valor "Maior que 10 saio da prisão!", e o resultado aberto da segunda rodada no modo em lote tem o valor (3 6) para a consulta `list/2/resultadoAberto`.

Os gráficos abaixo mostram os resultados da execução dos jogos da Tabela 5.2, exibindo certas métricas, como o tempo médio e o desvio padrão em segundos para a execução de uma jogada, o tempo médio em segundos e o desvio padrão necessário para realizar as operações de criptografia em uma jogada, e também o tráfego médio e o desvio padrão em *bytes* das mensagens trafegadas para executar uma jogada[5]. Num cenário com 2, 3, 4 e 5 *peers* no modo normal e em lote, as Figuras 5.1, 5.2 e 5.3 mostram o tempo médio de execução de uma jogada, e as Figuras 5.4, 5.5 e

[1]http://rpol.net
[2]http://www.wizards.com/dnd/
[3]http://web.archive.org/web/20040803073535/www.white-wolf.com/vampire
[4]http://paizo.com/pathfinderRPG
[5]Os resultados são apresentados em detalhes no Apêndice A.1.

Tabela 5.3: Resultados das jogadas comuns

Jogada	Modo	Jogador	Resultado fechado	Resultado Aberto
ND1	Normal	Alice	2	(2)
ND1	Lote	Bob	3	(3)
ND2	Normal	Alice	3	(2 1)
ND2	Lote	Carol	9	(3 6)
ND3	Normal	Alice	7	(2 1 4)
ND3	Lote	Carol	11	(5 5 1)
ND4	Normal	Alice	12	(2 5 5 1)
ND4	Lote	Alice	15	(3 8 4 3)
ND5	Normal	Carol	4	(4)
ND5	Lote	Alice	7	(7)
ND6	Normal	Alice	12	(8 4)
ND6	Lote	Bob	17	(9 8)
ND7	Normal	Alice	80	(80)
ND7	Lote	Alice	28	(28)
ND8	Normal	Bob	12	(12)
ND8	Lote	Alice	9	(9)

5.6 mostram o tempo médio das operações de criptografia, e as Figuras 5.7, 5.8 e 5.9 mostram a quantidade média de *bytes* trafegados entre os *peers* na execução de uma jogada.

Nota-se que para os gráficos dos jogos *Dungeons & Dragons* e *Pathfinder*, os resultados para o modo normal e para o modo em lote apresentam uma pequena diferença. No entanto, para os resultados no modo normal e no modo em lote para o jogo *Vampire The Masquerade*, a diferença é bem relevante. Isso decorre da quantidade de dados presentes em uma única jogada. Por exemplo, as notações predominantes no jogo *Vampire The Masquerade* são 8d10, 9d10, 6d10, enquanto nos outros jogos as notações mais comuns apresentam um baixo número de dados por jogada, como 1d20 + 13, 2d4, 2d10. Dessa forma, o modo normal apresenta um resultado bastante inferior quando o número de dados presentes na mesma jogada é elevado.

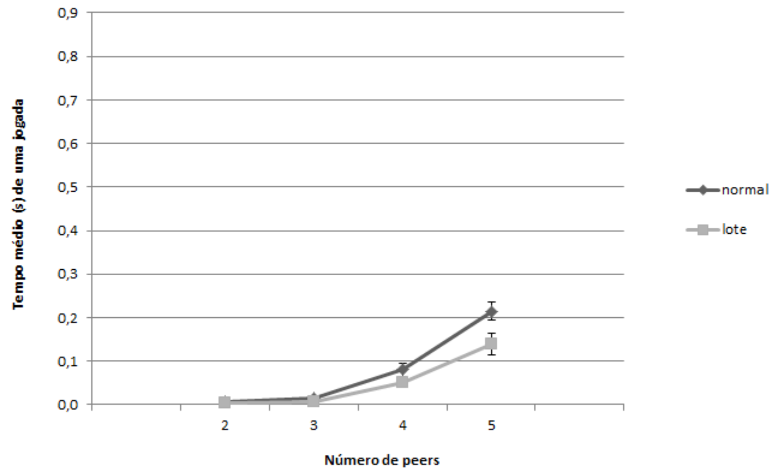

Figura 5.1: Tempo médio de jogada no jogo *Dungeons & Dragons*

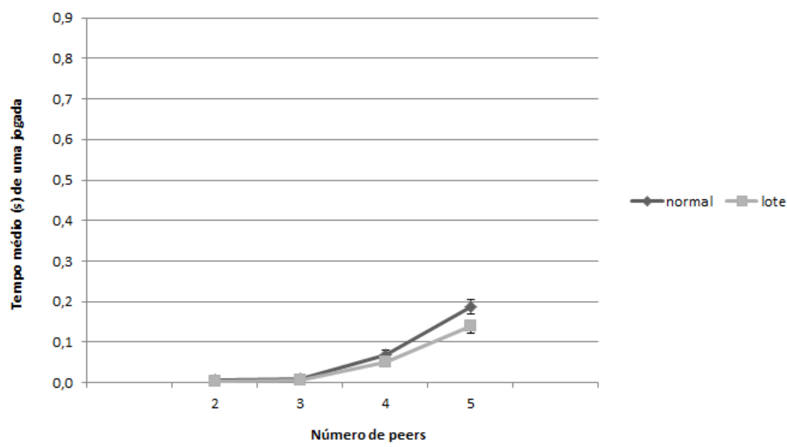

Figura 5.2: Tempo médio de jogada no jogo *Pathfinder*

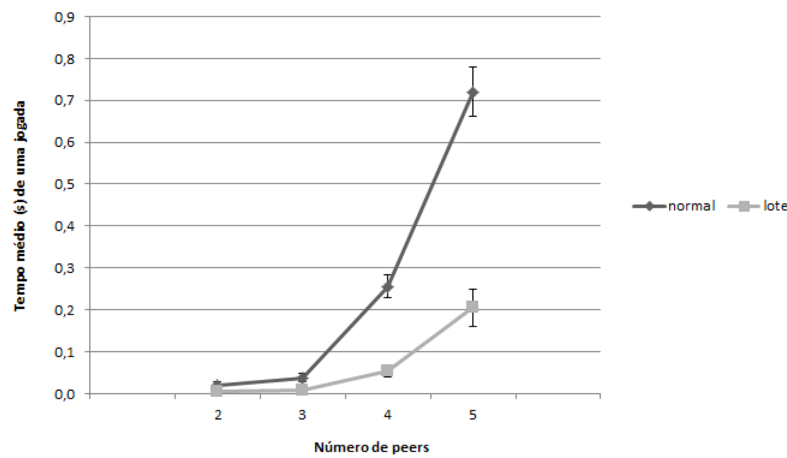

Figura 5.3: Tempo médio de jogada no jogo *Vampire The Masquerade*

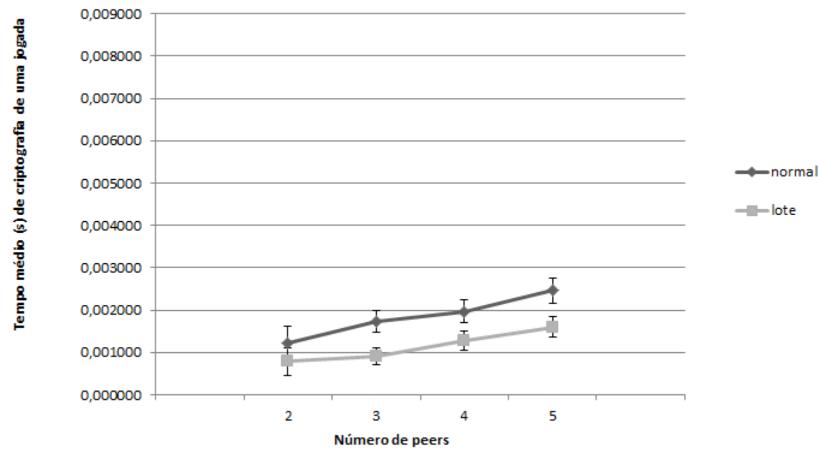

Figura 5.4: Tempo médio de criptografia por jogada no jogo *Dungeons & Dragons*

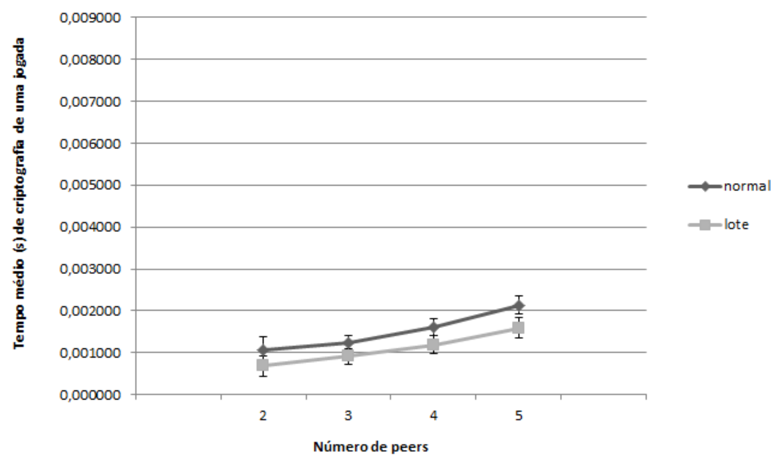

Figura 5.5: Tempo médio de criptografia por jogada no jogo *Pathfinder*

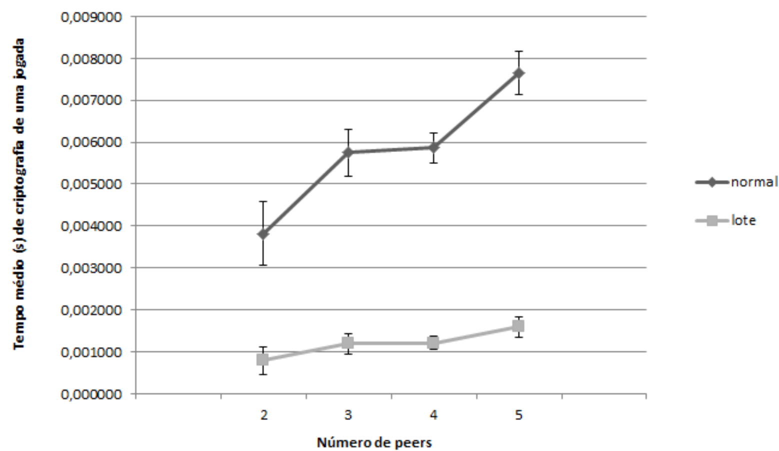

Figura 5.6: Tempo médio de criptografia por jogada no jogo *Vampire The Masquerade*

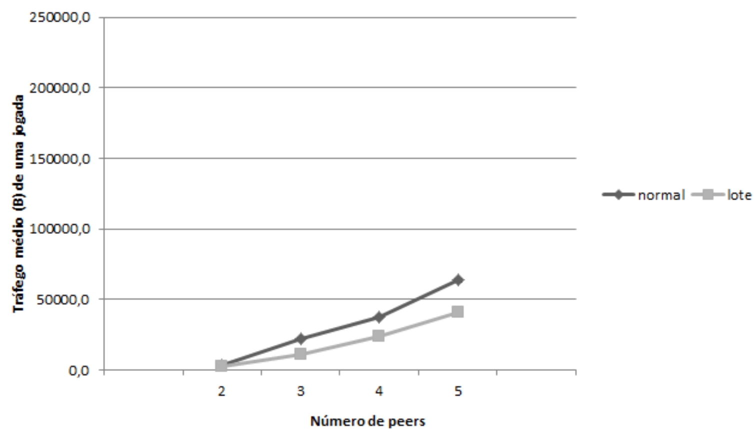

Figura 5.7: Tráfego médio por jogada no jogo *Dungeons & Dragons*

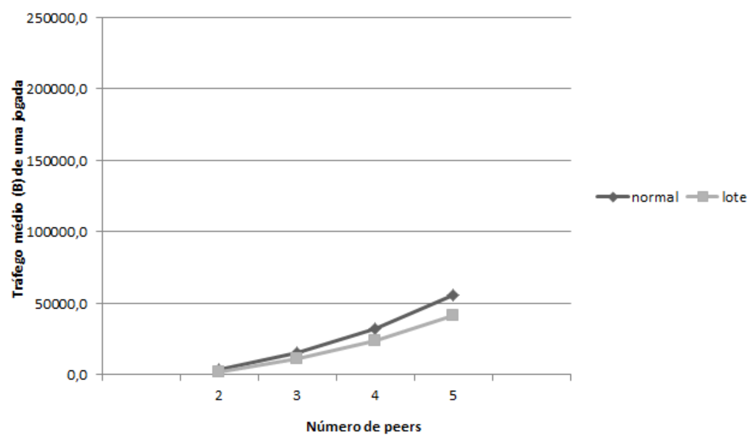

Figura 5.8: Tráfego médio por jogada no jogo *Pathfinder*

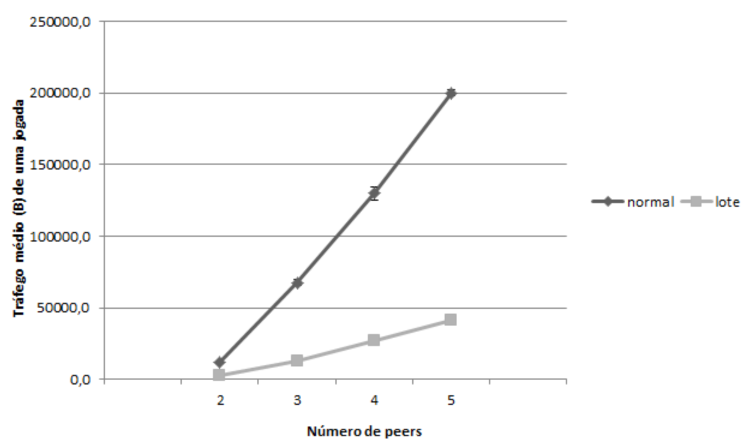

Figura 5.9: Tráfego médio por jogada no jogo *Vampire The Masquerade*

5.3 Experimento no módulo *Mental Poker*

O objetivo é testar a implementação do módulo *Mental Poker* variando o tipo de jogo e a quantidade de jogadores participantes. De fato, por se tratar da primeira implementação do protocolo proposto por GOLLE (2005), é de interesse avaliar algumas métricas do protocolo, como o tráfego médio em *bytes* por jogada, o número médio de colisões por jogada, o tempo médio necessário para executar as operações de criptografia em uma jogada e o tempo médio para completar uma jogada. A seguir é apresentado o cenário do experimento e os resultados obtidos.

5.3.1 Cenário do experimento

Assim como no experimento do módulo *Dice Rolling*, o experimento desse módulo é composto por *peers* que são aplicações console executando localmente. Cada *peer* está escutando em uma porta e está conectado com todos os outros *peers*. A aplicação console apresenta os seguintes comandos:

- `closedcard`: gera uma carta fechada para o jogador;
- `opencard`: gera uma carta aberta para o jogador;
- `opencardtotable`: gera uma carta aberta para a mesa;
- `hand`: retorna uma tabela que mostra as cartas que estão na mão do jogador. Cada linha da tabela contém o índice que é utilizado em outros comandos, o valor da carta criptografado e o valor da carta;
- `peerhand` > `peername`: retorna a mão do outro jogador. O parâmetro `peername` identifica o jogador. Exemplo: `peerhand` > `alice`;
- `tablecards`: retorna as cartas da mesa;
- `insertclosedcard` > `index`: insere uma carta da mão de volta ao baralho. O parâmetro `index` identifica a carta da mão do jogador;
- `discardopencard` > `index`: descarta uma carta de forma aberta. O parâmetro `index` identifica a carta da mão do jogador;
- `discardclosedcard` > `index`: descarta uma carta de forma fechada. O parâmetro `index` identifica a carta da mão do jogador;
- `getclosedcard` > `peername` > `index`: escolhe e pega uma carta da mão do outro jogador. O parâmetro `peername` identifica o jogador, e o parâmetro `index` identifica a carta da mão do jogador;

- `receiveclosedcard` $>$ `peername`: recebe uma carta da mão do outro jogador. O parâmetro `peername` identifica o jogador. Exemplo: `receiveclosedcard` $>$ `alice`;

Para obter as medidas de interesse, foram escolhidos quatro jogos para o experimento: jogo da Maior Carta, jogo de Pôquer, jogo de Sete e Meio e Escopa. O primeiro jogo, jogo da Maior Carta, é bem simples. Cada jogador retira uma carta aberta do baralho, e ganha quem retirar a maior carta. O segundo jogo, jogo de Pôquer, especificamente a variação chamada *Texas hold'em*, tem o seguinte procedimento: inicialmente, cada jogador recebe duas cartas fechadas, e então os jogadores fazem a primeira rodada de aposta. Três cartas abertas são geradas para a mesa. Outra rodada de aposta ocorre. Outra carta aberta é gerada para a mesa, seguida por outra rodada de aposta. Por fim, mais uma carta aberta é gerada para a mesa, e ocorre a última rodada de aposta. Nesse experimento, há o interesse somente nas jogadas do pôquer, e não nas rodadas de aposta. No terceiro jogo, jogo de Sete e Meio, os jogadores devem tentar somar sete e meio com as cartas da sua mão ou tentar chegar o mais próximo possível, mas sempre evitando ultrapassar o valor de sete e meio. Nesse jogo é utilizado um baralho de 40 cartas. Inicialmente, cada jogador recebe uma carta fechada. Em seguida, cada jogador pode comprar do baralho quantas cartas quiser para tentar chegar o mais próximo do valor sete e meio. No quarto e último jogo, Escopa, o baralho é composto por 40 cartas. Os jogadores recebem 3 cartas fechadas a cada rodada. Na primeira rodada também são geradas 4 cartas abertas para compor a mesa. Ao fim de cada rodada todos os jogadores jogaram as suas 3 cartas na mesa. Um jogo com 3 jogadores tem no máximo 2 rodadas.

Para os três primeiros jogos foram executadas 100 partidas, variando o número de jogadores entre 2, 3, 4 e 5 jogadores. Para o jogo Escopa utilizou-se 2 e 3 jogadores. Mais ainda, foi executado mais 100 jogos para o jogo de Maior Carta utilizando a abordagem proposta neste trabalho que evita colisões num jogo com cartas abertas. Com isso será possível medir o ganho de performance dessa abordagem. Também foi executado mais 100 jogos para o jogo Escopa com a abordagem proposta neste trabalho que evita gerar na rodada seguinte as cartas que foram abertas nas rodadas anteriores. Dessa forma, foram executadas 400 partidas para cada um dos três primeiros jogos, 200 partidas para o jogo de Escopa, 400 partidas para o jogo de Maior Carta sem colisão e também mais 200 partidas para o jogo de Escopa que evita colisão com as cartas que já foram geradas em rodadas anteriores. Portanto, o número total de partidas no experimento é 2.000 partidas. Como forma de automatizar a execução das partidas, foi necessário a participação de um *peer* com o papel de maestro, ou seja, que coordenasse as jogadas de uma partida dependendo do tipo de jogo. Esse *peer* maestro envia mensagens para os jogadores de forma que

estes executem as jogadas de acordo com a regras do jogo. Assim que um jogador termina sua jogada, o *peer* maestro é notificado e outro jogador assume a jogada.

5.3.2 Resultados

Os gráficos abaixo mostram os resultados[6] da execução das 2.000 partidas do experimento para os quatro tipos de jogos: jogo da Maior Carta, jogo de Pôquer, jogo de Sete e Meio e Escopa. As Figuras 5.10, 5.11, 5.12 e 5.13 apresentam o tráfego médio e o desvio padrão em *bytes* utilizado por jogada. As Figuras 5.14, 5.15, 5.16 e 5.17 apresentam o número médio e o desvio padrão de colisões por jogada. As Figuras 5.18, 5.19, 5.20 e 5.21 apresentam o tempo médio e o desvio padrão para as operações de criptografia por jogada. A Figura 5.22 apresenta o tempo médio e o desvio padrão para pegar uma carta aberta do baralho. A Figura 5.23 apresenta o tempo médio e o desvio padrão para gerar uma carta aberta do baralho para a mesa. Finalmente, as Figuras 5.24, 5.25 e 5.26 apresentam o tempo médio e o desvio padrão para pegar uma carta fechada do baralho.

As Figuras 5.10, 5.11, 5.12 e 5.12 mostram que os valores de tráfego médio por jogada são semelhantes nos quatro jogos. Interessante perceber que mesmo o tráfego médio do jogo Maior Carta seja semelhante aos outros jogos, seu desvio padrão é muito menor, pois comparando a média de colisões nas Figuras 5.14, 5.15, 5.16 e 5.16, o jogo de Maior Carta tem a menor média de colisão entre os jogos, ou seja, menos mensagens são trocadas e menos *bytes* são trafegados. Por outro lado, nos jogos de Pôquer e Sete e Meio o desvio padrão aumenta à medida que o número de jogadores cresce, pois o número médio de colisões também está aumentando. Da mesma forma, o jogo Escopa apresenta o maior desvio padrão entre todos os jogos, pois a colisão é muito maior devido à quantidade maior de cartas geradas no jogo de Escopa em relação aos outros jogos.

A Tabela 5.4 apresenta os resultados, em porcentagem, do aumento de performance na execução do jogo de Maior Carta com o protocolo sem colisão. Como para 2 jogadores não houve colisão, não houve nenhuma melhoria. Porém, para 3, 4 e 5 jogadores, houve uma melhora significativa que aumentou à medida em que o número médio de colisão foi maior. Da mesma forma, a Tabela 5.5 apresenta os resultados do aumento de performance na execução do jogos Escopa no modo que evita colisão com as cartas que foram geradas nas rodadas anteriores.

[6] Os resultados são apresentados em detalhes no Apêndice A.2.

Tabela 5.4: Aumento de performance no jogo de Maior Carta sem colisão

Métrica / Jogadores	2	3	4	5
Tráfego médio	0%	1,15%	1,17%	1,21%
Tempo médio de criptografia	0%	4,43%	6,50%	7,08%
Tempo médio para gerar carta aberta	0%	6,20%	7,93%	8,15%

Tabela 5.5: Aumento de performance no jogo Escopa no modo sem colisão com as rodadas anteriores

Métrica / Jogadores	2	3
Tráfego médio	0,99%	4,60%
Colisão	1,38%	6,38%
Tempo médio de criptografia	2,07%	5,30%
Tempo médio para gerar carta fechada	2,54%	6,49%

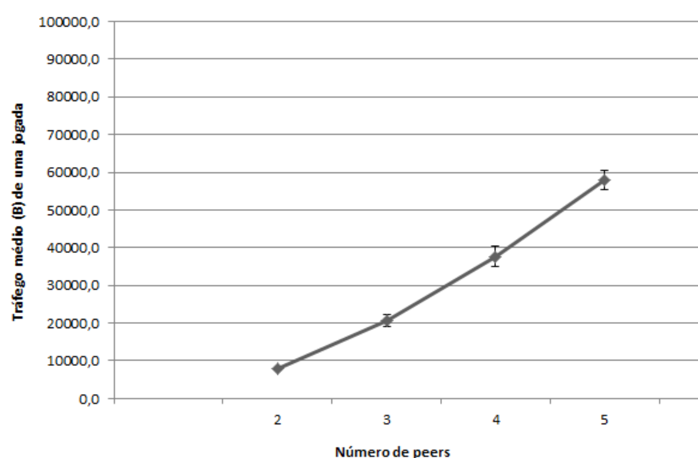

Figura 5.10: Tráfego médio por jogada no jogo de Maior Carta

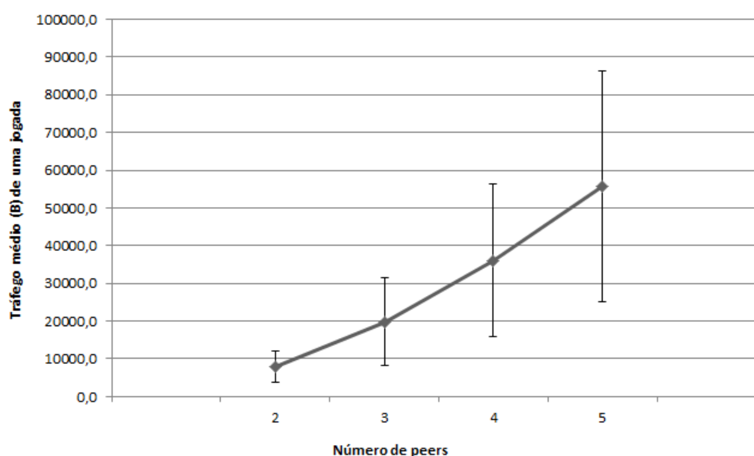

Figura 5.11: Tráfego médio por jogada no jogo de Poker

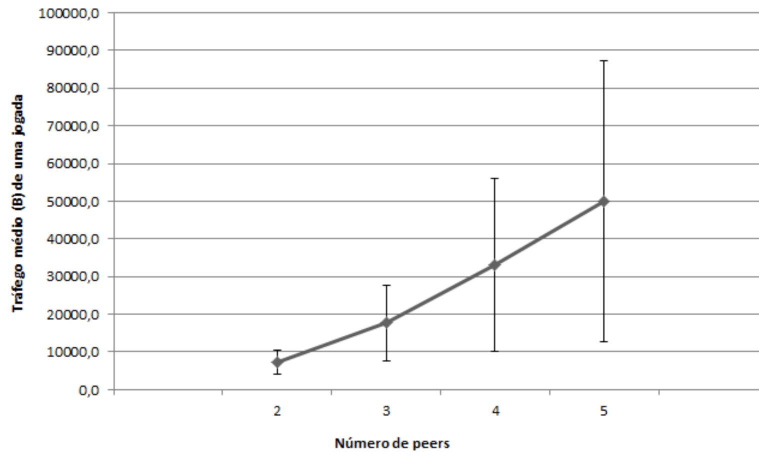

Figura 5.12: Tráfego médio por jogada no jogo de Sete e Meio

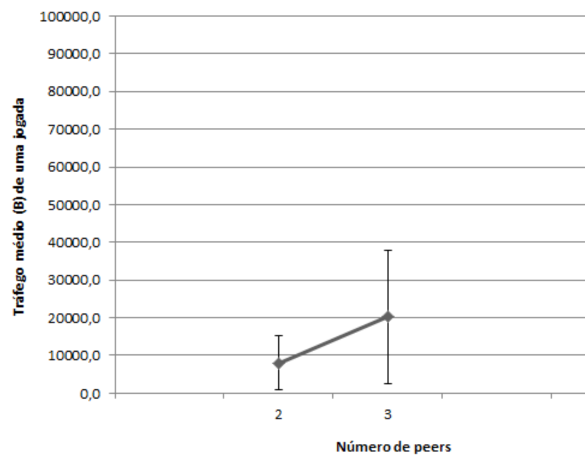

Figura 5.13: Tráfego médio por jogada no jogo Escopa

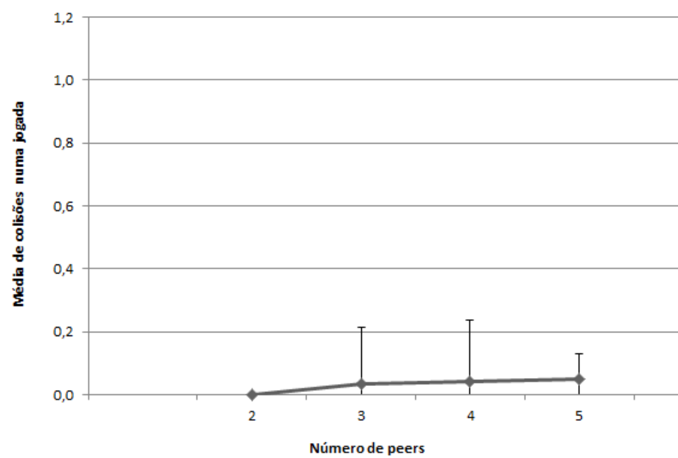

Figura 5.14: Média de colisão por jogada no jogo de Maior Carta

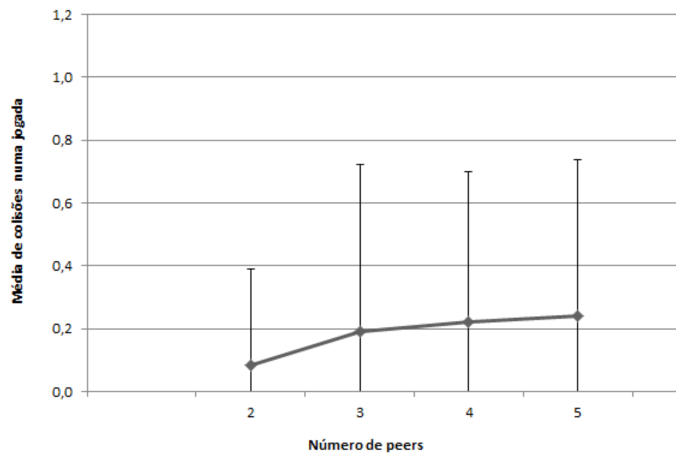

Figura 5.15: Média de colisão por jogada no jogo de Poker

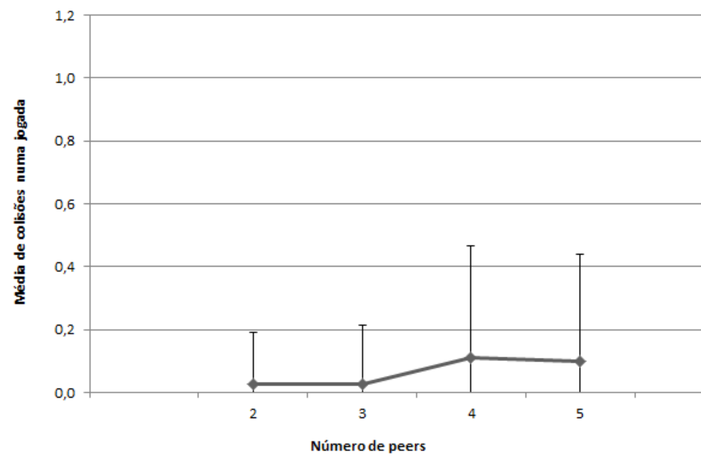

Figura 5.16: Média de colisão por jogada no jogo de Sete e Meio

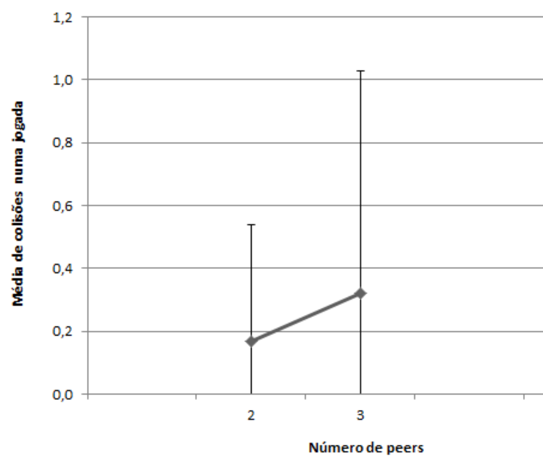

Figura 5.17: Média de colisão por jogada no jogo Escopa

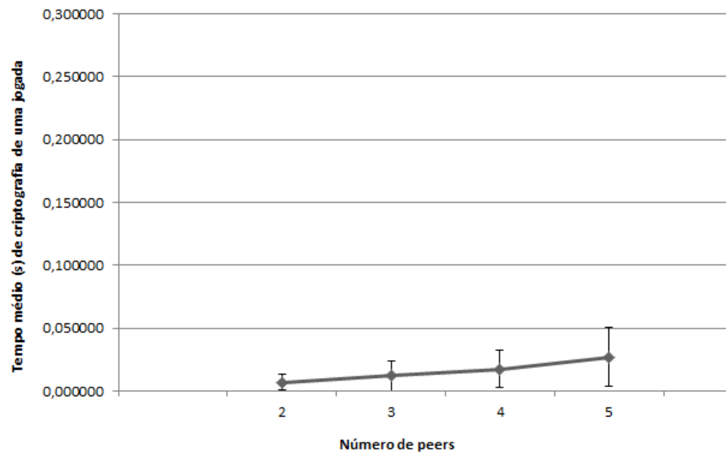

Figura 5.18: Tempo médio de criptografia por jogada no jogo de Maior Carta

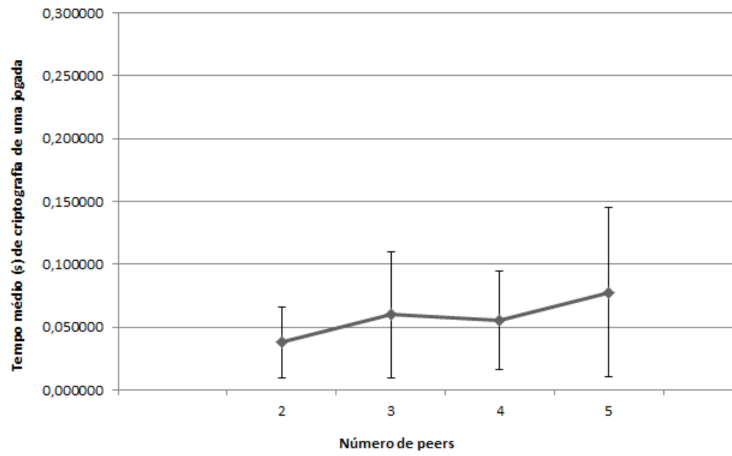

Figura 5.19: Tempo médio de criptografia por jogada no jogo de Poker

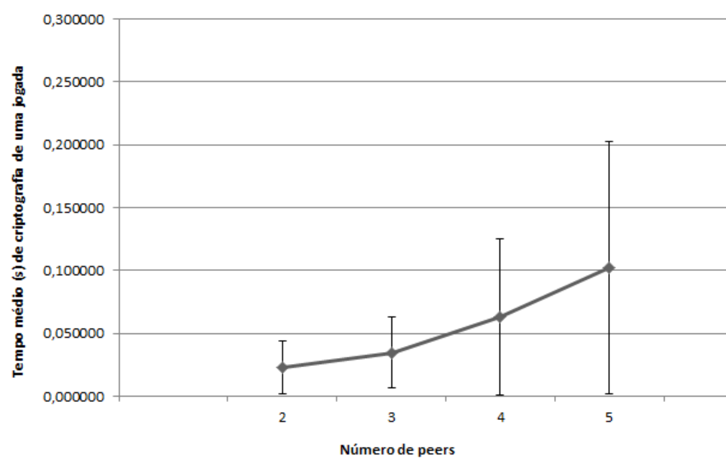

Figura 5.20: Tempo médio de criptografia por jogada no jogo de Sete e Meio

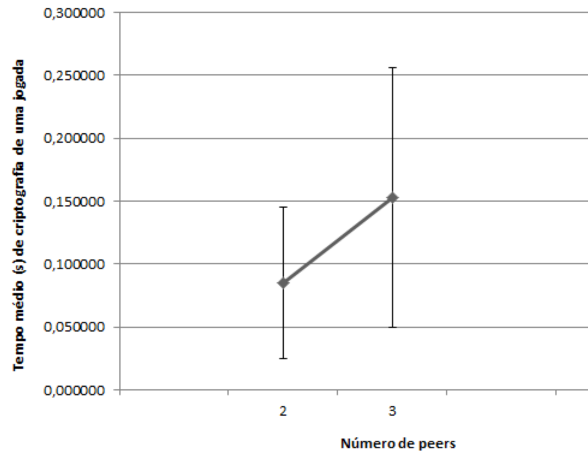

Figura 5.21: Tempo médio de criptografia por jogada no jogo Escopa

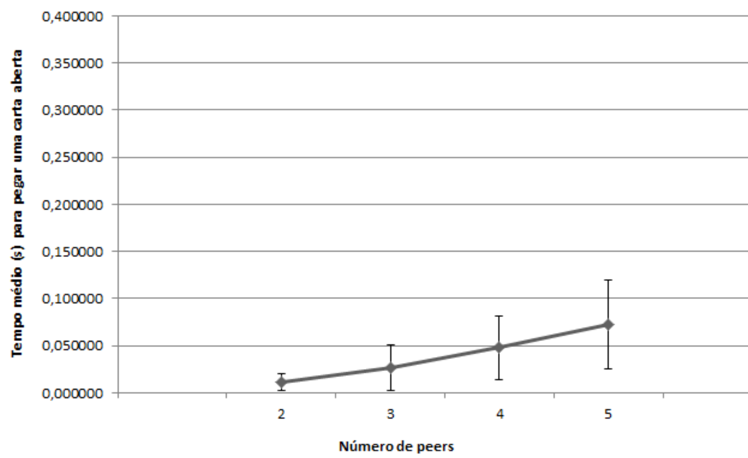

Figura 5.22: Tempo médio para gerar uma carta aberta por jogada no jogo de Maior Carta

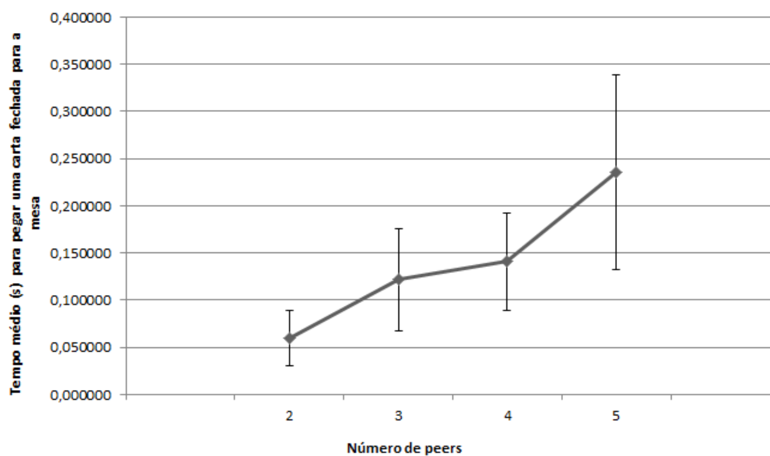

Figura 5.23: Tempo médio para gerar uma carta aberta na mesa por jogada no jogo de Poker

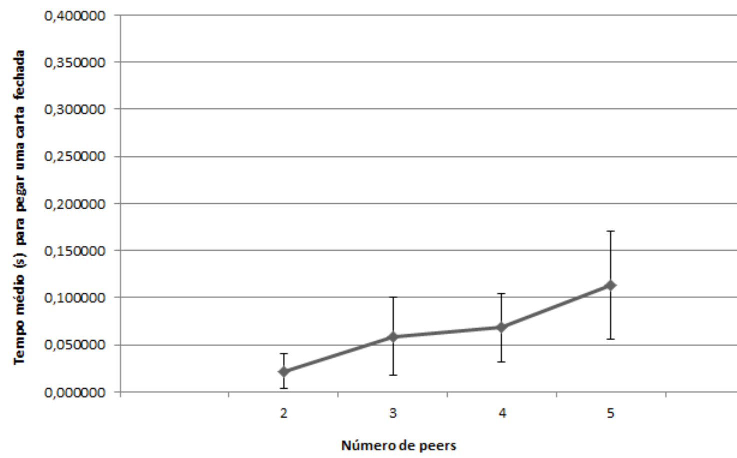

Figura 5.24: Tempo médio para gerar uma carta fechada por jogada no jogo de Poker

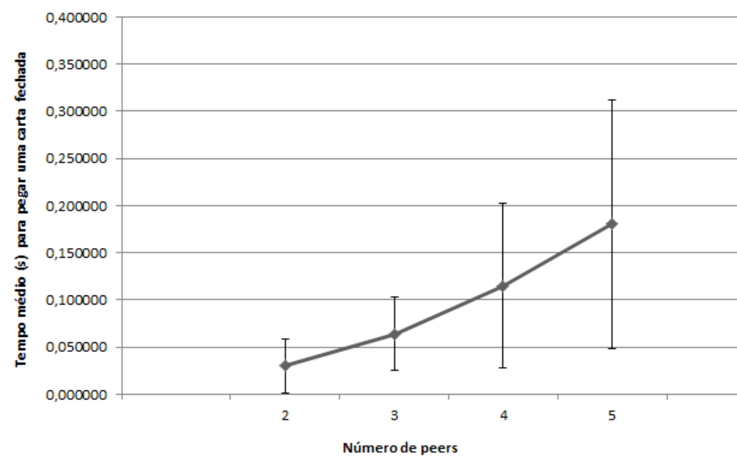

Figura 5.25: Tempo médio para gerar uma carta fechada por jogada no jogo de Sete e Meio

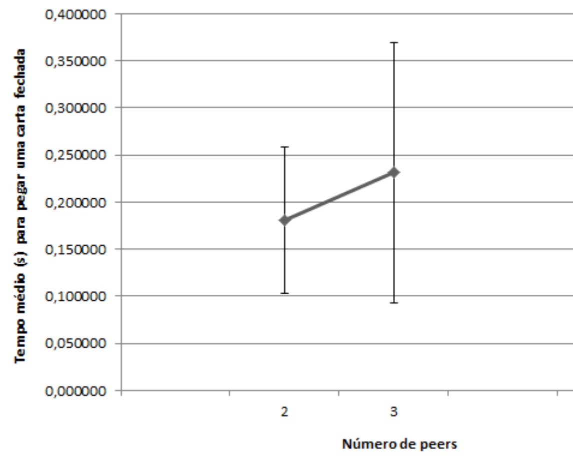

Figura 5.26: Tempo médio para gerar uma carta fechada por jogada no jogo Escopa

Capítulo 6

Conclusão

Este capítulo aborda as considerações finais referentes a este trabalho, suas contribuições e perspectivas futuras.

6.1 Considerações acerca do trabalho

Este trabalho apresentou um estudo sobre criptografia, especificamente sobre protocolos criptográficos que são utilizados para resolver um determinado problema entre dois ou mais participantes. Mais ainda, este trabalho propôs uma biblioteca que implementa um conjunto de protocolos criptográficos de suporte a ações em jogos num ambiente ponto-a-ponto.

Protocolos criptográficos estão relacionados à comunicação entre participantes e que têm o objetivo de prover segurança. Como alguns exemplos de uso de um protocolo criptográfico estão o protocolo de autenticação em um sistema; protocolo de definição das chaves utilizadas em uma sessão de comunicação; protocolo de particionamento e distribuição de um segredo entre os participantes; protocolo para jogar par ou ímpar, moeda ou dados; protocolo de jogar cartas, entre outros. No entanto, mesmo possuindo potencial em diferentes tipos de aplicação, existe uma carência de bibliotecas que documentam, implementam e distribuam protocolos criptográficos que endereçam ações em jogos em um ambiente ponto-a-ponto.

Para tal, este trabalho propôs uma biblioteca de suporte a ações em jogos em um ambiente ponto-a-ponto, precisamente ações relacionadas a jogos de dados e de cartas. A biblioteca foi composta por dois módulos chamados *Dice Rolling* e *Mental Poker*. O primeiro módulo provê uma interface simples de usar que permite dois ou mais jogadores executarem um cálculo conjunto e distribuído para jogar dados. Além de permitir gerar os dados a partir de uma notação de dados e também consultar qualquer resultado de um histórico de jogadas, o usuário desse módulo pode estendê-lo para interpretar outras notações de dados que o próprio usuário define e também permite sobrescrever o próprio protocolo criptográfico com outra implementação.

O segundo módulo, por meio de uma interface também simples de usar, provê a capacidade dos jogadores executarem em conjunto diversas operações relacionadas a um jogo de cartas, como pegar uma carta aberta ou fechada do baralho, jogar uma carta na mesa, descartar uma carta, entre outras. De fato, a implementação deste módulo é a primeira implementação conhecida do protocolo de cartas proposto por GOLLE (2005). Além disso, este trabalho também propôs e implementou diversas operações que estendem o protocolo de GOLLE (2005).

A biblioteca proposta neste trabalho é descrita em detalhes por meio de modelos e das decisões de engenharia de *software* utilizadas para modelar a sua arquitetura. Mais ainda, foi conduzida uma análise de complexidade dos algoritmos de cada operação. Por fim, foram realizados experimentos utilizando a biblioteca proposta a fim de analisar o desempenho de cada operação quando o número de jogadores aumenta. De fato os resultados estão de acordo com a análise de complexidade realizada.

6.2 Contribuições

A primeira contribuição deste trabalho é a biblioteca proposta para dar suporte a ações que envolvem sorte em jogos em um ambiente ponto-a-ponto. As contribuições deste trabalho podem ser listadas a partir de cada módulo da biblioteca:

Módulo *Dice Rolling*:

- Proposta de protocolo para jogar dados com n jogadores;
- Cálculo da complexidade da operação de jogar dado;
- Proposta do modo de jogar dados em lote para reduzir o custo da operação;
- Especificação do módulo da biblioteca por meio de modelo de dados e de diagrama de sequências;
- Implementação do módulo da biblioteca especificado;
- Explicação do uso do módulo da biblioteca e também dos possíveis pontos de extensão do módulo;
- Avaliação da operação de jogar dado no modo normal e em lote por meio da simulação de jogos reais com variação do número de jogadores.

Módulo *Mental Poker*:

- Primeira implementação conhecida do protocolo de GOLLE (2005), incluindo também a reimplementação de todos os outros protocolos e algoritmos utilizados pelo protocolo de GOLLE (2005), como o protocolo de igualdade de

texto de JAKOBSSON e SCHNORR (1999), o protocolo de geração distribuída de chaves de PEDERSEN (1991) e o algoritmo de chave pública ELGAMAL (1985);

- No trabalho de GOLLE (2005) não há uma verificação do protocolo de igualdade de texto de JAKOBSSON e SCHNORR (1999) para testar se uma carta é igual a outra. Dessa forma, este trabalho descreve em detalhes, incluindo prova matemática, de como o protocolo de igualdade de texto funciona e porque é utilizado para testar se uma carta é igual a outra sem revelar as cartas;

- Explicação, na prática (seção 4.3), do funcionamento e como deve ser implementado o algoritmo de igualdade de texto de JAKOBSSON e SCHNORR (1999) aplicado ao protocolo de GOLLE (2005);

- Proposta e implementação de várias operações uilizadas em jogos reais que estendem o protocolo de GOLLE (2005), como: jogadores geram uma carta para a mesa de forma aberta, jogador joga carta de sua mão na mesa de forma aberta, jogador coloca uma carta de sua mão novamente no baralho, jogador P_i escolhe e pega carta da mão do jogador P_j, jogador P_i recebe carta da mão do jogador P_j que foi escolhida pelo próprio jogador P_j, jogador descarta carta de sua mão no monte de descarte de forma aberta ou fechada, jogadores incluem cartas do monte de descarte de novo no baralho;

- Cálculo da complexidade de cada operação proposta;

- Cálculo da complexidade de cada operação do protocolo de GOLLE (2005);

- Proposta de eliminar a colisão do protocolo de GOLLE (2005) quando o jogo contém somente cartas abertas;

- Proposta que reduz o número de colisões do protocolo de GOLLE (2005) quando as cartas são abertas durante as rodadas de um jogo, permitindo o uso de menos cartas a cada rodada;

- Especificação do módulo da biblioteca por meio de modelo de dados e de diagrama de sequências;

- Implementação do módulo da biblioteca especificado;

- Explicação do uso do módulo da biblioteca;

- Avaliação de cada operação proposta e implementada por meio da simulação de jogos reais com variação do número de jogadores, verificando as complexidades calculadas.

6.3 Limitações e trabalhos futuros

De fato a biblioteca de suporte a ações em jogos ponto-a-ponto proposta neste trabalho pode ser melhorada, ampliada e aprimorada. Para o módulo *Mental Poker*, um trabalho futuro seria desenvolver alguma abordagem para evitar a etapa que, por meio do algoritmo de teste de igualdade de texto, verifica se uma carta já foi gerada durante o jogo. De fato esse passo eleva o protocolo de geração de cartas de uma complexidade $O(n^2)$ para $O(n^3)$, então eliminar o teste de igualdade de texto aumenta consideravelmente o desempenho das operações. Para um jogo que utiliza somente cartas abertas, este trabalho conseguiu propor uma abordagem para eliminar o passo que verifica se uma carta já foi gerada. No entanto, esta proposta é limitada, pois se existirem cartas fechadas no jogo, a proposta não funciona. Portanto, eliminar o teste de igualdade de texto em um jogo que utiliza cartas fechadas é um desafio que requer melhorias no protocolo ou até mesmo a utilização de outros protocolos criptográficos.

Como outro trabalho futuro está a ampliação da biblioteca para abranger mais tipos de ações que podem ser utilizadas em jogos ponto-a-ponto, como o leilão e o gerenciamento de recursos finitos. Para ações envolvendo leilão, o jogo Ra[1], por exemplo, utiliza práticas de leilão como mecanismo pelo qual os jogadores decidem em adquirir determinados itens durante o jogo. O trabalho de FRANKLIN e REITER (1996) apresenta uma proposta de protocolo distribuído que endereça ações em um leilão com o objetivo de garantir que as apostas somente sejam reveladas ao fim do período de apostas, que a casa de leilão tem a garantia de receber o pagamento do vencedor e que somente o vencedor consiga resgatar o item leiloado. O protocolo também provê proteção tanto para a casa de leilão quanto para os apostadores contra trapaças de outros apostadores maliciosos. As ações de gerenciamento de recursos finitos garantem, por exemplo, que em um jogo composto por 10 diamantes, nenhum jogador possua mais de 10 diamantes, ou que a soma dos diamantes que os jogadores possuem não seja maior que 10 diamantes. O protocolo também deve garantir que nenhum jogador consiga trapacear criando mais recursos indevidamente, mas deve permitir que recursos sejam trocados entre os jogadores. O protocolo também deve identificar e punir o jogador que tentar utilizar o mesmo recurso mais de uma vez. O conceito mais próximo desse tipo de protocolo é definido como dinheiro virtual ou dinheiro eletrônico[2] descrito detalhadamente em CHAUM (1982) e CHAUM (1988).

[1]http://boardgamegeek.com/boardgame/12/ra
[2]Tradução para o termo em inglês *digital cash*

Referências Bibliográficas

AL-OQILY, I., KARMOUCH, A., 2007, "Policy-Based Context-Aware Overlay Networks". In: *Global Information Infrastructure Symposium, 2007. GIIS 2007. First International*, pp. 85 –92, july. doi: 10.1109/GIIS.2007.4404172.

BARNETT, A., SMART, N. P., 2003, "Mental Poker Revisited." In: Paterson, K. G. (Ed.), *IMA Int. Conf.*, v. 2898, *Lecture Notes in Computer Science*, pp. 370–383. Springer. ISBN: 3-540-20663-9.

BLAKLEY, G., 1979, "Safeguarding cryptographic keys". In: *Proceedings of the 1979 AFIPS National Computer Conference*, pp. 313–317, Monval, NJ, USA. AFIPS Press.

BLUM, M., 1982, "Coin Flipping by Telephone - A Protocol for Solving Impossible Problems." In: *COMPCON*, pp. 133–137. IEEE Computer Society. Disponível em: <`http://dblp.uni-trier.de/db/conf/compcon/compcon1982.html#Blum82`>.

BRASSARD, G., CHAUM, D., CRÉPEAU, C., 1988, "Minimum Disclosure Proofs of Knowledge." *J. Comput. Syst. Sci.*, v. 37, n. 2, pp. 156–189. Disponível em: <`http://dblp.uni-trier.de/db/journals/jcss/jcss37.html#BrassardCC88`>.

CASTRO, M., DRUSCHEL, P., GANESH, A. J., et al., 2002, "Secure Routing for Structured Peer-to-Peer Overlay Networks." In: Culler, D. E., Druschel, P. (Eds.), *OSDI*. USENIX Association. ISBN: 978-1-4503-0111-4. Disponível em: <`http://dblp.uni-trier.de/db/conf/osdi/osdi2002.html#CastroDGRW02`>.

CHAUM, D., 1982, "Blind Signatures for Untraceable Payments." In: *Crypto*, v. 82, pp. 199–203.

CHAUM, D. L., 1988. "Blind signature systems". jul. 19. US Patent 4,759,063.

COMMUNITIES, E., 2012, *Rolling Dice.* Disponível em: <http://www.crockford.com/ec/Chap6RollingDice.html>. Acesso em: 9 de set. 2012.

COUTINHO, S. C., 2003, *Números Inteiros e Criptografia RSA.* 2 ed. , Instituto Nacional de Matemática Pura e Aplicada, IMPA. ISBN: 8524401249.

CRAMER, R., GENNARO, R., SCHOENMAKERS, B., 1997, "A Secure and Optimally Efficient Multi-Authority Election Scheme." In: Fumy, W. (Ed.), *EUROCRYPT*, v. 1233, *Lecture Notes in Computer Science*, pp. 103–118. Springer. ISBN: 3-540-62975-0.

CRÉPEAU, C., 1985, "A Secure Poker Protocol that Minimizes the Effect of Player Coalitions." In: Williams, H. C. (Ed.), *CRYPTO*, v. 218, *Lecture Notes in Computer Science*, pp. 73–86. Springer. ISBN: 3-540-16463-4.

CRÉPEAU, C., 1986, "A Zero-Knowledge Poker Protocol That Achieves Confidentiality of the Players' Strategy or How to Achieve an Electronic Poker Face." In: Odlyzko, A. M. (Ed.), *CRYPTO*, v. 263, *Lecture Notes in Computer Science*, pp. 239–247. Springer.

CRÉPEAU, C., 2012, *Commitment.* Disponível em: <http://crypto.cs.mcgill.ca/~crepeau/PDF/Commit.pdf>. Acesso em: 4 de ago. 2012.

DELFS, H., KNEBL, H., 2002, *Introduction to Cryptography: Principles and Applications.* Springer. ISBN: 3-540-42278-1.

DIFFIE, W., HELLMAN, M. E., 1976, "New directions in cryptography." *IEEE Transactions on Information Theory*, v. 22, n. 6, pp. 644–654. Disponível em: <http://dblp.uni-trier.de/db/journals/tit/tit22.html#DiffieH76>.

EDWARDS, J., 1994, *Implementing Electronic Poker: A Practical Exercise in Zero-Knowledge Interactive Proofs.* Tese de Mestrado, Department of Computer Science, University of Kentucky.

ELGAMAL, T., 1985, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", *CRYPTO*, v. IT-31, n. 4 (July), pp. 469–472.

FORTUNE, S., MERRITT, M., 1984, "Poker Protocols." In: Blakley, G. R., Chaum, D. (Eds.), *CRYPTO*, v. 196, *Lecture Notes in Computer Science*, pp. 454–464. Springer. ISBN: 3-540-15658-5.

FRANKLIN, M. K., REITER, M. K., 1996, "The design and implementation of a secure auction service", *Software Engineering, IEEE Transactions on*, v. 22, n. 5, pp. 302–312.

FREEMAN, A., JONES, A., 2003, *Programming .NET Security*. 1 ed. , O'Reilly Media. ISBN: 0596004427.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1995, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-63361-2.

GOLDREICH, O., LEVIN, L. A., 1989, "A Hard-Core Predicate for all One-Way Functions". In: Johnson, D. S. (Ed.), *STOC*, pp. 25–32. ACM. ISBN: 0-89791-307-8. Disponível em: <http://dblp.uni-trier.de/db/conf/stoc/stoc89.html#GoldreichL89>.

GOLDWASSER, S., MICALI, S., 1982, "Probabilistic encryption & how to play mental poker keeping secret all partial information". In: *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pp. 365–377. ACM.

GOLLE, P., 2005, "Dealing Cards in Poker Games." In: *ITCC (1)*, pp. 506–511. IEEE Computer Society. Disponível em: <http://dblp.uni-trier.de/db/conf/itcc/itcc2005-1.html#Golle05>.

HARDY, G. G. H., WRIGHT, E. M., 1979, *An Introduction to the Theory of Numbers*. Oxford University Press.

IRELAND, K., ROSEN, M., 1990, *A Classical Introduction to Modern Number Theory*, v. 84. Springer.

JAKOBSSON, M., SCHNORR, C.-P., 1999, "Efficient Oblivious Proofs of Correct Exponentiation". In: *Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks: Communications and Multimedia Security*, CMS '99, pp. 71–86, Deventer, The Netherlands, The Netherlands. Kluwer, B.V. ISBN: 0-7923-8600-0.

JGAMMON, 2012, *Biblioteca JGammon*. Disponível em: <http://sourceforge.net/projects/jgam/#screenshots>. Acesso em: 9 de set. 2012.

KIPM, 2013, *Standard Dice Notation*. Disponível em: <http://homepage.ntlworld.com/dice-play/Notation.htm>. Acesso em: 6 de abril 2013.

KUBIATOWICZ, J., 2003, "Extracting guarantees from chaos." *Commun. ACM*, v. 46, n. 2, pp. 33–38. Disponível em: <http://dblp.uni-trier.de/db/journals/cacm/cacm46.html#Kubiatowicz03>.

LIPTON, R., 1981, "How to Cheat at Mental Poker". Proceedings of the AMS Short Course in Criptography.

LUA, E. K., CROWCROFT, J., PIAS, M., et al., 2005, "A survey and comparison of peer-to-peer overlay network schemes." *IEEE Communications Surveys and Tutorials*, v. 7, n. 1-4, pp. 72–93. Disponível em: <http://dblp.uni-trier.de/db/journals/comsur/comsur7.html#LuaCPSL05>.

MENEZES, A., V. OORSHOT, P., VANSTONE, S., 1996, *Handbook of Applied Cryptography*. CRC Press.

MEYER, B., 1988, *Object-Oriented Software Construction, 1st edition.* Prentice-Hall. ISBN: 0-13-629031-0.

MILLER, G. L., 1976, "Riemann's hypothesis and tests for primality", *Journal of computer and system sciences*, v. 13, n. 3, pp. 300–317.

NACCACHE, D., STERN, J., 1997, "A new public-key cryptosystem". In: *Advances in Cryptology—EUROCRYPT'97*, pp. 27–36. Springer.

NAOR, M., 1991, "Bit Commitment Using Pseudorandomness." *J. Cryptology*, v. 4, n. 2, pp. 151–158. Disponível em: <http://dblp.uni-trier.de/db/journals/joc/joc4.html#Naor91>.

NAPSTER, 2012, *Napster*. Disponível em: <http://www.napster.com>. Acesso em: 10 de jun. 2012.

NIVEN, I., ZUCKERMAN, H. S., MONTGOMERY, H. L., 2008, *An Introduction to the Theory of Numbers*. John Wiley & Sons.

ORAM, A., 2001, *Peer-to-peer: Harnessing the benefits of a disruptive technology.* Sebastopol, CA, O'Reilly.

PAILLIER, P., 1999, "Public-key cryptosystems based on composite degree residuosity classes". In: *Advances in cryptology—EUROCRYPT'99*, pp. 223–238. Springer.

PEDERSEN, T. P., 1991, "A Threshold Cryptosystem without a Trusted Party (Extended Abstract)." In: Davies, D. W. (Ed.), *EUROCRYPT*, v. 547, *Lecture Notes in Computer Science*, pp. 522–526. Springer. ISBN: 3-540-54620-0.

RABIN, M. O., 1980, “Probabilistic algorithm for testing primality”, *Journal of Number Theory*, v. 12, n. 1, pp. 128 – 138. ISSN: 0022-314X.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., et al., 2001, “A scalable content-addressable network”. In: *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 161–172. ACM.

RISSON, J., MOORS, T., 2007. “Survey of Research towards Robust Peer-to-Peer Networks: Search Methods”. RFC 4981 (Informational), set. Disponível em: <http://www.ietf.org/rfc/rfc4981.txt>.

RIVEST, R. L., ADLEMAN, L., DERTOUZOS, M. L., 1978a, “On data banks and privacy homomorphisms”, *Foundations of secure computation*, v. 32, n. 4, pp. 169–178.

RIVEST, R. L., SHAMIR, A., ADLEMAN, L. M., 1978b, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.” *Commun. ACM*, v. 21, n. 2, pp. 120–126. Disponível em: <http://dblp.uni-trier.de/db/journals/cacm/cacm21.html#RivestSA78>.

ROUSSOPOULOS, M., BAKER, M., ROSENTHAL, D. S. H., et al., 2003, “2 P2P or Not 2 P2P?” *CoRR*, v. cs.NI/0311017.

ROWSTRON, A., DRUSCHEL, P., 2001, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, v. 11, pp. 329–350.

SCHINDELHAUER, C., 1998, *A Toolbox for Mental Card Games.* Relatório técnico, University of Lubeck.

SCHNEIER, B., 1996, *Applied cryptography - protocols, algorithms, and source code in C (2. ed.).* Wiley. ISBN: 978-0-471-11709-4.

SCHOLLMEIER, R., 2001, “A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications.” In: Graham, R. L., Shahmehri, N. (Eds.), *Peer-to-Peer Computing*, pp. 101–102. IEEE Computer Society. ISBN: 0-7695-1503-7. Disponível em: <http://dblp.uni-trier.de/db/conf/p2p/p2p2001.html#Schollmeier01>.

SHAMIR, A., 1979, “How to Share a Secret.” *Commun. ACM*, v. 22, n. 11, pp. 612–613. Disponível em: <http://dblp.uni-trier.de/db/journals/cacm/cacm22.html#Shamir79>.

SHAMIR, A., RIVEST, R. L., ADLEMAN, L. M., 1979. "Mental Poker". Technical Report MIT-LCS-TM-125, Massachusetts Institute of Technology.

STAMER, H., 2013, *SecureSkat. Uma implementação criptograficamente segura do jogo Skat.* Disponível em: <http://savannah.nongnu.org/projects/secureskat/>. Acesso em: 7 de julho de 2013.

STAMER, H., 2005, "Efficient Electronic Gambling: An Extended Implementation of the Toolbox for Mental Card Games." In: Wolf, C., Lucks, S., Yau, P.-W. (Eds.), *WEWoRC*, v. 74, *LNI*, pp. 1–12. GI. ISBN: 3-88579-403-9.

STEINMETZ, R., WEHRLE, K., 2005, *Peer-to-peer systems and applications*. Berlin [u.a.], Springer. ISBN: 3-540-29192-X.

STOICA, I., MORRIS, R., KARGER, D., et al., 2001, "Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications", *Computer Communication Review*, v. 31, n. 4 (out.), pp. 149–160.

TEBAA, M., EL HAJJI, S., EL GHAZI, A., 2012, "Homomorphic Encryption Applied to the Cloud Computing Security". In: *Proceedings of the World Congress on Engineering*, v. 1.

WALLACH, D. S., 2002, "A Survey of Peer-to-Peer Security Issues." In: Okada, M., Pierce, B. C., Scedrov, A., et al. (Eds.), *ISSS*, v. 2609, *Lecture Notes in Computer Science*, pp. 42–57. Springer. ISBN: 3-540-00708-3. Disponível em: <http://dblp.uni-trier.de/db/conf/isss2/isss2002.html#Wallach02>.

WANG, C., LI, B., 2003, *Peer-to-Peer Overlay Networks: A Survey*. Relatório técnico.

WINTER, S., 2007, *A Brief History of Dice*. Disponível em: <http://www.wizards.com/default.asp?x=dnd/alumni/20070302a>. Acesso em: 6 de abril 2013.

ZHAO, B., KUBIATOWICZ, J., JOSEPH, A., 2001, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing", *Computer*, v. 74.

Apêndice A

Resultados do experimento

A.1 *Dice Rolling*

Tabela A.1: Resultados das métricas no jogo *Dungeons & Dragons* no modo normal

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,006530884	0,013994751	0,079799609	0,2144569880
σ do tempo médio (s)	0,00461839	0,006322008	0,015130991	0,0211467824
Tempo médio de criptografia (s)	0,00121727	0,002535283	0,001966699	0,002457099
σ do tempo médio de criptografia (s)	0,000406531	0,0002569	0,000273228	0,000294198
Tráfego médio (B)	3734,039168	22663,62497	37260,91723	63698,85404
σ do tráfego médio (B)	178,1331278	135,5823034	854,0635073	1189,303457
Total de mensagens	4	20	42	72

Tabela A.2: Resultados das métricas no jogo *Dungeons & Dragons* no modo em lote

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,004238889	0,007122888	0,051794163	0,1391939175
σ do tempo médio (s)	0,003753315	0,004343218	0,012296792	0,0245454332
Tempo médio de criptografia (s)	0,000790073	0,000914734	0,001276491	0,001594787
σ do tempo médio de criptografia (s)	0,000330383	0,000202357	0,00022205	0,000239092
Tráfego médio (B)	2423,588732	11551,03266	24184,30421	41343,92222
σ do tráfego médio (B)	144,7668544	449,4158267	694,0881178	966,5339763
Total de mensagens	4	20	42	72

Tabela A.3: Resultados das métricas no jogo *Pathfinder* no modo normal

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,005729893	0,010386213	0,068797757	0,1873652562
σ do tempo médio (s)	0,003536699	0,004502086	0,011615463	0,0182771320
Tempo médio de criptografia (s)	0,001067976	0,001231611	0,001615602	0,002137564
σ do tempo médio de criptografia (s)	0,000311315	0,000185288	0,000205603	0,000216393
Tráfego médio (B)	3276,071978	15700,54451	32560,69146	55844,92296
σ do tráfego médio (B)	136,4118592	463,8009908	608,820068	879,238309
Total de mensagens	4	20	42	72

Tabela A.4: Resultados das métricas no jogo *Pathfinder* no modo em lote

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,004236111	0,007683564	0,050895545	0,1391601260
σ do tempo médio (s)	0,001835552	0,004777831	0,01232689	0,0166852960
Tempo médio de criptografia (s)	0,000680208	0,000911127	0,001195198	0,001594787
σ do tempo médio de criptografia (s)	0,000245941	0,000196636	0,000218196	0,000239092
Tráfego médio (B)	2409,16338	11615,02648	24087,90939	41343,92222
σ do tráfego médio (B)	134,495767	492,2080154	646,1092653	966,5339763
Total de mensagens	4	20	42	72

Tabela A.5: Resultados das métricas no jogo *Vampire The Masquerade* no modo normal

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,020508293	0,038141194	0,255129483	0,7203376849
σ do tempo médio (s)	0,008493845	0,00959858	0,027021272	0,0578665116
Tempo médio de criptografia (s)	0,003822475	0,005749246	0,005749246	0,007649461
σ do tempo médio de criptografia (s)	0,000747665	0,005491224	0,005491224	0,00051619
Tráfego médio (B)	11725,63613	67564,37616	67564,37616	200046,2258
σ do tráfego médio (B)	327,6109404	2272,038157	2272,038157	2212,12194
Total de mensagens	4	20	42	72

Tabela A.6: Resultados das métricas no jogo *Vampire The Masquerade* no modo em lote

	2 peers	3 peers	4 peers	5 peers
Tempo médio (s)	0,004238889	0,007883459	0,052733083	0,2053505802
σ do tempo médio (s)	0,003753315	0,004241483	0,011940336	0,0446852960
Tempo médio de criptografia (s)	0,000790073	0,00118832	0,001211452	0,001594787
σ do tempo médio de criptografia (s)	0,000330383	0,002426498	0,00015571	0,000239092
Tráfego médio (B)	2423,588732	12775,43284	26821,00787	41343,92222
σ do tráfego médio (B)	144,7668544	1057,895845	2041,032036	966,5339763
Total de mensagens	4	20	42	72

A.2 *Mental Poker*

Tabela A.7: Resultados das métricas no jogo Maior Carta

	2 peers	3 peers	4 peers	5 peers
Tempo médio para gerar carta aberta (s)	0,01109049	0,026415405	0,047748037	0,072606976
σ do tempo médio para gerar carta aberta (s)	0,008931798	0,024011464	0,03341112	0,047123501
Tempo médio de criptografia (s)	0,006642213	0,011943959	0,017433254	0,02703301
σ do tempo médio de criptografia (s)	0,00616958	0,011744748	0,015026583	0,023614288
Tráfego médio (B)	7880,285	20547,07383	37691,1	57930,12
σ do tráfego médio (B)	154,876124	11620,708454	2838,530873	2585,481288
Colisão média (B)	0	0,033557	0,04	0,0505
σ da colisão média	0	0,180086012	0,195959179	0,077226938

Tabela A.8: Resultados das métricas no jogo Pôquer

	2 peers	3 peers	4 peers	5 peers
Tempo médio para gerar carta fechada (s)	0,021803915	0,058754123	0,068273101	0,113370206
σ do tempo médio para gerar carta fechada (s)	0,01835094	0,041317747	0,036119103	0,05709661
Tempo médio para gerar carta aberta na mesa(s)	0,060105696	0,12176581	0,140861398	0,235610475
σ do tempo médio para gerar carta aberta na mesa (s)	0,02944398	0,054249242	0,051727173	0,103290351
Tempo médio de criptografia (s)	0,037948616	0,059734877	0,055415122	0,07770165
σ do tempo médio de criptografia (s)	0,028218048	0,050192575	0,039310466	0,067457178
Tráfego médio (B)	8023,502222	19814,64275	36036,80769	55593,52933
σ do tráfego médio (B)	4192,026523	11549,69346	20171,97828	30579,96354
Colisão média (B)	0,082222222	0,189189189	0,22	0,2406
σ da colisão média	0,305351447	0,534348923	0,478643918	0,494934159

Tabela A.9: Resultados das métricas no jogo Sete e Meio

	2 peers	3 peers	4 peers	5 peers
Tempo médio para gerar carta fechada (s)	0,029869	0,063784405	0,114987555	0,18032327
σ do tempo médio para gerar carta fechada (s)	0,028272747	0,038930024	0,086893326	0,13145472
Tempo médio de criptografia (s)	0,022867739	0,034692011	0,062937553	0,102335676
σ do tempo médio de criptografia (s)	0,021002652	0,028537259	0,062368056	0,100332421
Tráfego médio (B)	7306,815287	17650,8	33111,04134	49998,78392
σ do tráfego médio (B)	3215,181998	10180,9948	23031,34187	37208,18901
Colisão média (B)	0,027600849	0,025165563	0,109359606	0,098333333
σ da colisão média	0,163826257	0,187425409	0,356309883	0,342048567

Tabela A.10: Resultados das métricas no jogo Escopa

	2 peers	3 peers
Tempo médio para gerar carta fechada (s)	0,180733115	0,23159646
σ do tempo médio para gerar carta fechada (s)	0,078238796	0,138238796
Tempo médio de criptografia (s)	0,085270001	0,1527506
σ do tempo médio de criptografia (s)	0,060190098	0,103140484
Tráfego médio (B)	8021,787402	20202,71212
σ do tráfego médio (B)	7225,610588	17545,69404
Colisão média (B)	0,166144201	0,31986532
σ da colisão média	0,371513849	0,707530773