Dependable Public Announcement Service

João Tavares
86443
joaobernardo.28@gmail.com

Madalena Pedreira
86466
mena.pedreira@gmail.com

1. INTRODUÇÃO

Neste relatório propomo-nos a discutir a implementação e design de um Dependable Public Announcement Server (DPAS), fazendo alusão aos possíveis ataques, mecanismos de proteção e garantias de integridade e dependabilidade providenciadas pelo sistema. O sistema foi implementado em Java, com a utilização de Sockets Object Streams.

2. AMBIENTE DE EXECUÇÃO

Antes do sistema arrancar, o ficheiro Run.Java gera um par RSA de chaves pública e privada. De seguida, atribui de forma aleatória um desses pares a cada *Server*/*Client* e guarda-os na respectiva KeyStore. Essa KeyStore, por nós desenvolvida, guarda o par de chaves atribuído a cada entidade de forma segura - já que ambas as chaves são encriptadas em base64 e guardadas num ficheiro txt, sendo que a chave privada é cifrada usando a password do *Server*/*Client* respectivo.

3. IMPLEMENTAÇÃO E DESIGN

A implementação e design dos diversos comandos mantém-se bastante semelhante à da entrega anterior, tendo sido acrescentados mecanismos de prevenção de falhas Bizantinas.

Os protocolos Bizantinos usados contam com uma noção de tolerância a falhas de N > 3f. A restrição de N deve ser feita de acordo com a quantidade de falhas Bizantinas que o sistema deve tolerar. O nosso sistema conta com a tolerância a 1 falha, necessitando para o efeito, de 4 *Servers*.

3.1 Protocolo de Comunicação

Optámos por uma implementação através de sockets TCP. Deste modo, existe confiabilidade na entrega dos pacotes e flexibilidade na implementação da comunicação a estabelecer entre *Clients* e *Servers*.

3.2 Servidores

Um *Server* é executado num dos 4 portos disponíveis - no âmbito do projeto, disponibilizámos os portos 4000, 5000, 6000 e 7000. Ao ser executado, este terá de se registar para que os *Clients* se possam conectar. O registo passa pela escrita do porto num ficheiro global, consultado pelos vários intervenientes do sistema. O *Server* passará então a aguardar pela conexão dos *Clients*, criando neste evento uma thread responsável por o atender.

Todos os mecanismos tiveram por base o algoritmo Authenticated-Data Byzantine Quorum que implementa um (1,N) Byzantine Regular Register. Expandimos este algoritmo de acordo com as necessidades inerentes a cada mecanismo de post/read e postGeneral/readGeneral. Todos os pacotes bizantinos enviados entre *Server* e *Client* têm uma assinatura do sender. Nestes pacotes enviamos a *Message* e os respectivos readIDs and writeTimestamps.

3.2.1 Private Announcement Board

Esta implementação teve por base o algoritmo Read-Impose Write-Majority que implementa (1,N) Byzantine Atomic Registers.

Para garantirmos que cada faulty *Server* recupera os seus posts em falta, deveríamos ter implementado o algoritmo Byzantine Quorum with Listeners. Os *Servers* iriam manter uma lista de processos a executar operações de leitura concorrentes - os listeners. Deste modo, quando um *Server* recebesse um write request com um novo timestamp associado, ele iria enviar esse request a todos os listeners.

O algoritmo por nós implementado recupera apenas dos posts perdidos aquando de um read(x) e sempre dos posts perdidos após o último post guardado.

Numa operação de write, o escritor garante que uma maioria de processos adota o seu valor. Numa operação de read, um leitor seleciona o valor com maior timestamp de uma maioria e impõe esse valor e garante que uma maioria o adopta antes de completar a operação: o que é crucial para assegurar a propriedade de ordenação do atomic register.

3.2.1 General Board

Esta implementação teve por base o algoritmo Read-Impose Write-Consult-Majority que implementa (N,N) Byzantine Atomic Registers.

A operação de read é bastante semelhante à do algoritmo de single-writer e muita da sua implementação coincide com a da operação write, que passamos a descrever: Cada escritor começa por consultar todos os outros *Servers* - fase de consulta - determinando o maior timestamp escrito até ao momento, que irá incrementar e associar à sua operação de escrita. O algoritmo estende os timestamps atendendo ao rank do processo-escritor para determinar o valor a escrever - assumimos que o porto menor, terá o menor rank.

A propriedade de atomicidade está assegurada pelo uso de um quorum de maioria durante as operações de read e write, bem como pela ordenação de timestamp/rank.

3.3 Clientes

Após a ligação com o *Server* ser estabelecida, todos os input do utilizador são processados do lado do *Client*, criando uma *Message* - objeto que separa e guarda os vários parâmetros desse input - que envia para o *Server*. A implementação e design dos diversos comandos mantém-se bastante semelhante à da entrega anterior.

4. FALHAS E GARANTIAS

Falhas de input: Para cada pedido do *Client*, é feito o saneamento do input recebido a nível do *Server*.

Falhas de assinatura: Nunca é enviada por nenhuma das partes uma *Message* não assinada. Caso o *Client* não consiga assinar a *Message*, o sistema avisa o utilizador deste acontecimento. O Server, por sua vez, tenta voltar a assinar a sua *Message* até 3 vezes.

Falha de verificação de assinatura: Não sendo verificada a assinatura: o *Server* fecha a conexão com o *Client*. Se este erro surgir na parte do *Client*, este imprime apenas uma *Message* para o utilizador.

5. ATAQUES E GARANTIAS

Spoofing - Como consequência de cada *Message* ter tido o seu conteúdo passado por uma hash (SHA-256) e encriptado com a private key do sender (mecanismo encapsulado pelo objecto java.Security.SignedObject), passa a existir uma assinatura verificável.

Tampering - O sistema garante que o receiver de uma *Message* é alertado se o conteúdo desta for alterado, já que cada *Packet* contém os dados da *Message* passados por uma hash e posteriormente encriptados com a private key do sender. Qualquer alteração na *Message* acusaria uma hash diferente (há uma probabilidade muitíssimo reduzida de haver uma colisão dos dados na hash).

Non-Repudiation - Para cada *Message* é calculada a sua hash que é posteriormente encriptada com a private key do sender. Deste modo, é conseguida uma assinatura digital que autentica o sender. A criação de uma assinatura digital para outra pessoa é altamente improvável por não serem viáveis ataques de colisão a esta hash.

Replay Attacks - Cada *Message* trocada no sistema vem acompanhada com um *Nonce* do sender. O *Server* e o *Client* têm cada um uma estrutura de dados que guarda o último *Nonce* que recebeu da outra parte. Se voltar a receber o mesmo *Nonce*, a mensagem é repudiada - um *Nonce* não pode ser utilizado mais que uma vez.

Persistence - O sistema garante persistência de dados na escrita e caso exista um crash do *Server*. No primeiro caso, num pedido de *Post* por parte do *Client*, o *Server* escreve em dois ficheiros. O segundo ficheiro é temporário e serve apenas para prevenir uma escrita interrompida. No segundo caso, todas as estruturas de dados do *Server* são serializadas e guardadas num ficheiro - é feito o backup em todas as operações que alterem as estruturas de dados. Quando um *Server* é inicializado verifica primeiro se tem algum estado anterior guardado, carregando-o.

Atomic Persistence - O sistema utiliza estruturas de dados concorrentes, garantindo que execuções paralelas não mudam o seu estado concorrentemente - os mecanismos de atomicidade das estruturas previnem isso.

6. CLIENTES BIZANTINOS

Considerando a possibilidade de um *Client* ser Bizantino, este poderia ignorar operações, enviar valores diferentes para diferentes processos, retornar valores arbitrários, etc. Lidar com a enorme diversidade de ataques possíveis é de extrema complexidade. No entanto, iremos passar a descrever uma solução para lidar com o envio de mensagens, por parte do *Client*, com conteúdo diferente para diferentes *Servers*.

Solução: Implementação de um Byzantine Reliable Broadcast, nomeadamente do algoritmo Authenticated Double-Echo Broadcast. Para $N > 3f$, o algoritmo garante que se um processo correcto entregar *m*, então qualquer processo correcto entrega também *m*, independemente de se o sender é, ou não, faulty.

Inicialmente, o *Client* envia uma *Message* m para todos os N *Servers*. Assim que um *Server* recebe essa *Message* m, retransmite ("echo") essa *Message* a todos os N *Servers*. Quando um *Server* tiver recebido um quorum Bizantino desses echoes - isto é, #echoes > (N + f) / 2 - envia uma mensagem READY a todos os *Servers*, informando da sua disponibilidade para entregar a resposta (de forma "reliable") ao *Client*. Assim que um *Server* tiver recebido 2f + 1 READYs, entrega a resposta ao *Client*. Caso o *Server* tenha recebido apenas f + 1 READYs e ainda não enviou a sua mensagem READY, envia-a para todos os *Servers*. Este passo de amplificação das mensagens READY é crucial para assegurar a propriedade de totality do algoritmo.

Garantias: Se um processo correcto entregar a *Message*, então recebeu no mínimo 2f + 1 mensagens READY, das quais pelo menos f +1 de processos correctos. Essas f + 1 irão eventualmente ser entregues a todos os processos correctos, despoletando o passo da amplificação e, consequentemente, mensagens suficientes para assegurar totality. Em relação às restantes garantias, o algoritmo por nós implementado de Authenticated Echo Broadcast já garantia validity, no-duplication e integrity. A sua garantia de consistency implica, também, que se alguns dos processos correctos enviarem uma mensagem READY, todos o fazem com o mesmo conteúdo *m*. Assim, não é possível que o processo faulty introduza mensagens READY suficientes com um conteúdo diferente de *m*.