TÉCNICO LISBOA

HDS Notary

Relatório - Projeto (Parte 2)

Trabalho realizado por:
Ruben Condesso, nº 81969
André Mendes, nº78079
João Bernardino, nº78022

1. Introdução

Este sistema denominado por *Highly Dependable Systems* (HDS) *Notary* é uma aplicação, altamente confiável, como o nome indica, cujo o seu objetivo é certificar a transferência de bens arbitrários entre os utilizadores existentes na aplicação. A primeira parte do projeto incidia sobre em desenvolver o sistema com as respetivas funcionalidades, e colocar mecanismos de segurança para que não houvesse formas ou meios de adulterar a informação do sistema. No entanto, nesta fase existia apenas um servidor sendo que, como era dito no enunciado, toda a informação recebida do mesmo era confiável.

Na segunda parte do projeto, passou a existir várias réplicas do servidor e a informação recebida pelas mesmas, deixou de ser confiável. O objetivo era , então, ampliar as implementações da primeira parte, para o sistema conseguisse tolerar faltas provenientes tanto dos servidores como dos clientes. As especificações do sistema permaneceram inalteradas, mas agora, o sistema teria de ser resiliente a faltas bizantinas provenientes dos servidores e/ou clientes.

2. Arquitetura Geral do Sistema

Neste ponto será especificado e ilustrado o design do sistema, nomeadamente, quais os módulos que fazem parte do sistema e o seu propósito geral na aplicação, e finalmente será explicado como está implementado o módulo da comunicação entre os servidores e clientes.

Design do Sistema:

O sistema é constituído, principalmente, pelas classes *Notary* e *Client*. A primeira mantém toda a informação relativa à aplicação, nomeadamente, executar as funcionalidades que a aplicação contém (vender e transferir bens, etc), e principalmente, encarregar-se de validar todas as operações feitas no sistema. A classe *Client* permite a interação entre cada utilizador e o(s) servidor(es), ou seja , é responsável por fazer chegar a cada servidor todas as ações que o cliente queira tomar, entre as quais vender e comprar bens.

Existem mais duas classes que complementam o sistema: a *MessageHandler* (*MessageHandler* e *MessageHandler1*), e a RSA. A classe *MessageHandler* tem como única função, definir o formato das mensagens trocadas na aplicação, enquanto a classe RSA tem a responsabilidade de gerar e armazenar as chaves RSA, e respetivos certificados, usados por cada interveniente existente no sistema.

A seguinte figura ilustra a arquitetura geral do nosso sistema, entre apenas um cliente e um servidor:

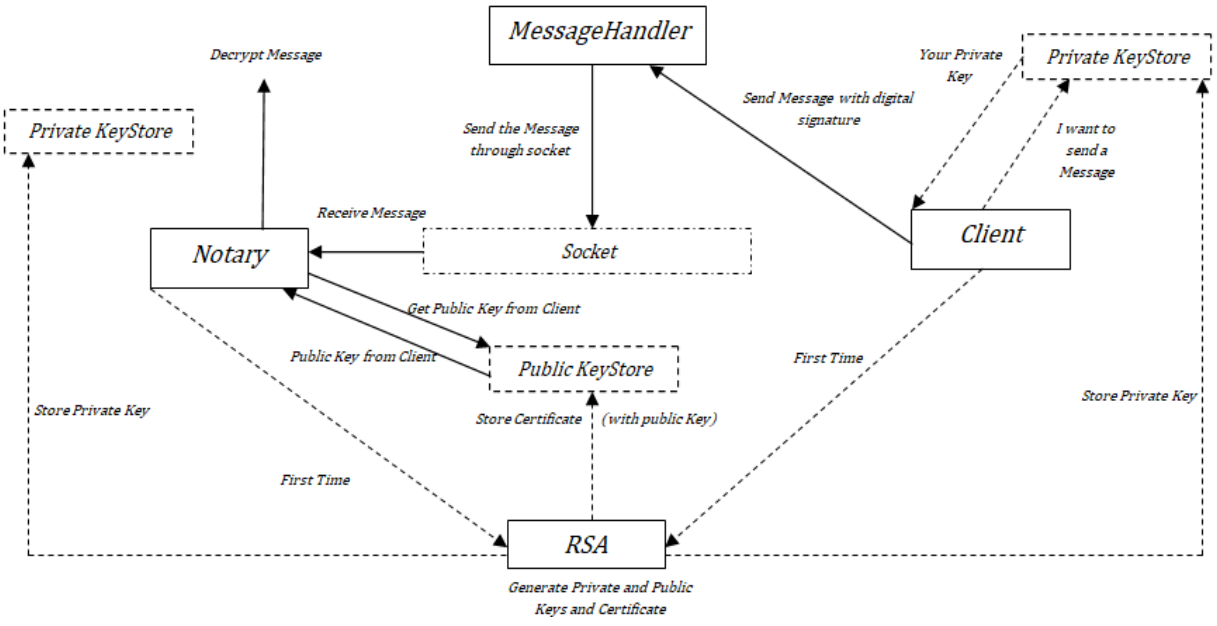

Figura 1 - Arquitetura do Sistema

A figura realça meramente a parte da segurança imposta no sistema, que será especificada e explorada nas secções seguinte. É visível ver a interação entre as várias classes do sistema, que foram brevemente explicadas nos parágrafos anteriores.

O número de clientes e servidores existente na aplicação é limitado, por questões de segurança, sendo esse limite definido por nós. De notar que o número de servidores tem de cumprir a seguinte condição: 3f + 1, sendo f o número de faltas que o sistema consegue tolerar. O propósito e o porquê desta condição será aprofundo na secção 3 do relatório.

Módulo de Comunicação:

Para cliente e servidor comunicar é utilizado *Sockets*, sob TPC, como mecanismo de transmissão das mensagens. Esta escolha deveu-se às seguintes razões:

- A primeira razão vai ao encontro de já termos usado este mecanismo noutros projetos ao longo do nosso percurso académico, e logo já estávamos familiarizados com o seu funcionamento;
- A segunda razão prendeu-se com o facto de termos concluído que as sockets, sob TCP, iriam cumprir corretamente o propósito do contexto do projeto, sendo que estas apresentavam-se como um mecanismo viável de transmissão de mensagens, entre um servidor e um cliente, cuja ligação é iniciada, mantida e terminada, ao longo do tempo;

As *sockets* sob TCP podem apresentar uma baixa velocidade de transmissão, mas neste projeto a fiabilidade terá um peso maior que a velocidade na hora de entregar as mensagens.

Dado que na aplicação vão existir vários servidores e clientes, terão de haver canais *sockets* existentes entre cada servidor e cada cliente, juntamente com canais *sockets* existentes entre todos os clientes, pois estes tem de comunicar entre si para haver uma transferência de bem. Para existir um paralelismo de ações (ou seja, ter a capacidade de fazer várias ações em simultâneo), cada interveniente da aplicação, clientes e servidores, serão *multi-threading*.

Funcionalidades do Sistema:

No enunciado é referido as várias operações que deverão ser implementadas no sistema, nomeadamente, a venda, compra e transferência de bens entre clientes. Todas essas operações foram corretamente implementadas no nosso sistema. Na secção 3, no primeiro ponto (Requisitos de Segurança), encontra-se explicado um pormenor específico relativo às operações de compra e transferência de bens, mas dado o teor desse pormenor faz mais sentido estar exposto nessa secção.

Numa operação (venda de um bem por exemplo), cada cliente irá receber as respostas dos servidores existentes na aplicação. No entanto, apenas uma resposta será mostrada ao cliente, e não as n respostas dos servidores. A resposta será escolhida consoante o resultado do algoritmo relativo ao registo atómico bizantino. Usando este algoritmo, cada cliente não será "obrigado" a esperar por todas as respostas dos servidores face a uma operação, e poderá tolerar f faltas bizantinas por parte dos servidores. Este tipo de procedimentos será explicado em detalhe na secção 4.

A figura seguinte ilustra, em parte, este exemplo:

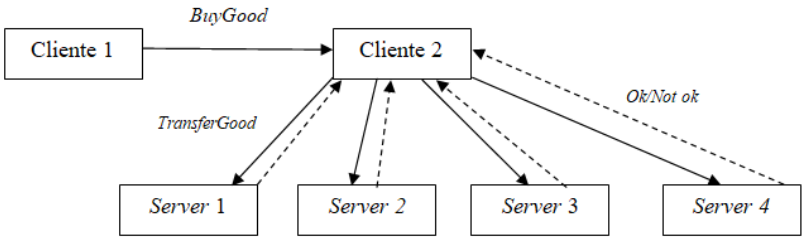

Figura 2 - Exemplo de funcionamento do Sistema numa compra/venda de bens

3. Imposições de Confiabilidade

Nesta secção é exposto quais foram os requisitos de confiabilidade que tinham de existir no nosso sistema, tanto em termos de segurança como e em termos de tolerância a faltas, dado o que era imposto no enunciado do projeto.

Requisitos de Segurança:

O enunciado do projeto deixa bem claro que os utilizadores não são confiáveis, pois podem atacar o sistema para próprio benefício ou simplesmente podem querer quebrar o normal funcionamento do sistema. Além disso, todos os pedidos de venda e compra de bens deverão ser não repudiáveis.

É imposto, também, que tem de existir uma total transparência nas mensagens trocadas no sistema, ou seja, qualquer cliente (existente na aplicação) poderá ler qualquer mensagem que passe pelo sistema. No entanto, as mensagens não poderão ser alteradas nem modificadas por nenhum interveniente, ou seja, tem de se garantir a integridade das mensagens.

Requisitos de Tolerância a faltas:

Relativamente à tolerância a faltas, o servidor tem de ter a capacidade de recuperar a informação existente na aplicação, caso haja um *crash* seu, independentemente da razão que levou a esse *crash*. Ou seja, a implementação do *Notary* deverá garantir que não existe perdas e/ou corrompimento do seu estado interno, caso existe um *crash*.

Se a parte da segurança incidia mais sobre o facto dos utilizadores não serem confiáveis, a parte da tolerância a faltas incide mais sobre o facto das respostas dos servidores também não serem confiáveis. Assim, é nos imposto que seja implementado, no nosso sistema, um (1, N) *Byzatine Atomic Register*, para existir um algoritmo que faça com que o sistema consiga tolerar faltas bizantinas, por parte dos servidores. É explicitado que certos métodos terão de ser considerados como operações de escrita e outros como operações de leitura. Este algoritmo é necessário porque, entre os servidores existentes, poderá haver alguns maliciosos, que ao receber as mensagens dos clientes poderão devolver respostas erradas voluntariamente.

No entanto, além dos servidores poderem ser maliciosos, os clientes também o podem ser. Logo, o sistema deverá ter implementando um algoritmo que também tenha isso em conta. Os casos mais sensíveis serão aqueles onde um cliente envia mensagens devidamente assinadas e com todos os requisitos necessários para a mensagem ser aceite pelos servidores, mas no entanto, o conteúdo da mensagem propriamente dita, difere. Um exemplo simples deste caso seria um cliente, na mesma "ronda" de mensagens, enviar um pedido para transferir o bem A para o servidor 1 e ao mesmo tempo estar a pedir o estado do bem B para o servidor 2. Ou seja, o teor das duas mensagens difere enquanto devia ser igual. Um algoritmo como o *Authenticated Echo Broadcast* resolveria este tipo de problemas.

Finalmente, é também imposto que haja um mecanismo de combate ao *spam*, por parte do(s) cliente(s) ao(s) servidor(es). Isso poderá ser feito, impondo um custo computacional no método referente à transferência de bens, no cliente, para assim diminuir o número transações que cada cliente consegue fazer num espaço de tempo.

4. Métodos implementados

De seguida, irá ser exposto os mecanismo implementados por nós para garantir os requisitos referidos na secção 3.

Mecanismos de Segurança:

Os mecanismos de segurança existentes no nosso sistema, têm como base o algoritmo criptográfico RSA. Este algoritmo é um tipo de encriptação assimétrica, portanto são geradas duas chaves: uma pública e outra privada; e isto é feito tanto no servidor como no cliente. As chaves públicas serão, portanto, conhecidas por todos os intervenientes da aplicação, e as chaves privadas serão conhecidas apenas pelos intervenientes em causa. Cada interveniente gera um certificado através da sua chave privada e pública, e irá guardar a chave privada, juntamente com o respetivo certificado, numa *keystore*. Desta forma, só o interveniente em causa consegue ter acesso à chave privada. As chaves públicas de todos os intervenientes são guardadas num único *keystore*.

Posto isto, aquando o envio de uma mensagem, é criado uma assinatura digital com a chave privada do interveniente em causa, e na receção da mesma mensagem, a assinatura recebida é verificada usando a chave pública do emissor. Em cada mensagem, irão estar contidos vários parâmetros, definidos na classe *MessageHandler*, que serão usados para o bom funcionamento do sistema. De notar, que estes parâmetros irão sempre protegidos com a assinatura digital. Sendo a integridade a propriedade que garante que a informação recebida mantenhas as características originais, ou seja, que esta não foi alterada ou modificada durante o processo de transferência da mensagem, usando este processo do uso da assinatura digital, com recurso a certificados e chaves RSA, esta propriedade é garantida.

As propriedades de autenticidade e não repúdio são também garantidas usando este processo de assinaturas digitais, pois a autenticidade diz à garantia que a informação da fonte anunciada, e que não foi alvo de alterações durante todo o processo, e o não repúdio previne as partes integrantes numa transação não possam negar uma transação após a sua realização. Como as mensagens são sempre assinadas pelo emissor, estas duas propriedades são garantidas.

Para garantir a frescura das mensagens, cada mensagem tem um tempo de expiração, e é enviado em cada mensagem o *timestamp* em que esta foi gerada, onde o recetor irá comparar esse valor recebido com o *timestamp* atual (quando recebeu a mensagem). Caso essa diferença seja superior a um tempo específico (por exemplo, 10 segundos), a mensagem não irá ser aceite. Além disto, o servidor e cada respetivo cliente, irão ter um número sequencial sincronizado entre ambos, que irá sendo incrementado à medida que cada um vá enviado e/ou recebendo mensagens. Este número irá ser enviado em cada mensagem, onde só irão ser aceites mensagens, cujo número sequencial recebido seja igual (ou maior) ao número sequencial que o recetor tem na altura.

É importante referir, que cada *Notary* irá verificar cuidadosamente todos os *inputs* recebidos dos clientes, sendo que é referido no enunciado que o servidor não pode confiar nos mesmos, pois estes podem atacar o sistema para benefício próprio. Irão ser detetados *inputs* anómalos e/ou *inputs* onde, claramente, o cliente está a tentar tirar proveito para ele próprio.

O uso das assinaturas digitais serviu, também, para contribuir para o bom funcionamento das operação da compra e transferência de bens. Como está ilustrado no enunciado do projeto, quando um cliente quer comprar um bem, irá comunicar diretamente com o dono desse bem, e este último irá comunicar com o(s) servidor(es) para validar a transferência. No entanto, neste processo, não existe comunicação entre o comprador e o servidor. Logo, o servidor não terá, à partida, nenhuma prova em como o comprador quis efetivamente compara aquele bem. Para contrariar este problema, a solução encontrada foi a seguinte: quando um cliente quer comprar um bem, notifica o vendedor envia-lhe uma mensagem onde se encontra o bem que quer comprar. Juntamente com essa mensagem, o comprador envia uma assinatura. O vendedor, ao enviar o pedido de transferência ao servidor, irá enviar a sua mensagem assinada mais a assinatura recebida pelo comprador. Assim, o servidor irá verificar a mensagem assinada pelo vendedor, mais a assinatura que é referente ao comprador. Se as duas assinaturas forem corretas, o servidor terá a confirmação em que os dois intervenientes quiserem estar, efetivamente, no negócio.

Mecanismos de Tolerância a Faltas:

No nosso contexto, denomina-se uma falta quando existiu um acontecimento que alterou o padrão normal de funcionamento de uma dada componente do sistema. E por conseguinte, uma falta irá dar origem a um erro e um erro irá dar origem a uma falha. Temos que considerar que essa falha poderá ter sido causa involuntariamente ou voluntariamente. Caso tenha sido causa de forma voluntária, temos de considerar que é uma falta arbitrária (ou bizantina), ou seja, o processo está a enviar uma reposta que não devia, e isso poderá dever-se a existir uma entidade maliciosa, existente na aplicação (cliente ou servidor), que estará a forjar as respostas. Vamos agora ilustrar os mecanismos existentes no nosso sistema, contra possíveis faltas que venham afetar o normal funcionamento do mesmo.

Vamos supor que um cliente tem um *crash*, seja porque foi fechado abruptamente seja porque sofreu um ataque que o levou a esse desfecho. O facto dos servidores serem *multi-thread* permite que não sejam afetados nestes casos, pois simplesmente fecham a *thread* relativa a esse cliente (e por conseguinte, a ligação), e dão continuidade ao normal funcionamento da aplicação.

No entanto, não é apenas o cliente que poderá sofrer um *crash* repentino, os servidores poderão sofrer o mesmo. Logo, para combater isto a informação referente aos bens existentes na aplicação e os respetivos donos, é guardada num ficheiro serializado, sendo que é atualizado sempre que existirem alterações neste contexto. Assim, caso exista u *crash* do servidor, este ao voltar a ser iniciado irá recuperar a informação que continha antes de ter sofrido esse *crash*. A escrita para este ficheiro é feita de forma sincronizada, usando *synchronized* (no JAVA), para garantir que não há erros de escrita para o ficheiro, caso haja várias escritas em simultâneo. Além deste ficheiro, é também criado um ficheiro que contém o histórico de transações do sistema, que irá sofrer o mesmo processo referido anteriormente.

Considerando agora a parte do enunciado, onde é dito que os servidores são, igualmente, não confiáveis, ou seja, poderão existir servidores bizantinos na nossa aplicação, cujo objetivo passe por forjar as respostas às operações, e assim, fornecer informação inválida aos clientes, acabando por pôr em causa o bom funcionamento da aplicação. É neste contexto, que foi necessário implementar o (1, N) *Byzantine Atomic Register*, referido no enunciado. Iremos, de seguida, expor de que forma está implementado o nosso algoritmo para contrapor estas situações.

O algoritmo utilizado segue o procedimento do *Byzantine Atomic Register*, do tipo *Authenticated-Data Byzantine Quorum*, usando *Authenticate Perfect Point to Point Links.* Logo toda a informação considerada neste algoritmo iria ser, sempre, devidamente assinada para depois ao ser recebida, o recetor possa verificar a mensagem recebida (ou seja, a sua integridade). Assim não é permitido falsificar mensagens originadas de outros processos. A primeira condição a ser imposta no algoritmo, era apenas considerar uma resposta por servidor, e fazer essa verificação, para prevenir eventuais casos, onde um servidor possa estar a enviar múltiplas respostas.

De seguida, dividiu-se o método de verificar o estado de um bem como operação de leitura, e os métodos de vender e transferir bem como operações de escrita. Na operação de leitura, todas as mensagem consideradas vêm assinadas (com assinatura digital), como foi referido anteriormente. O primeiro passo será verificar se a assinatura é válida. O passo seguinte será verificar o *rid* (id da operação) recebido na mensagem, para se ter a certeza que estamos sempre a considerar as respostas relativas às mesmas "perguntas". Caso estes dois passos sejam ultrapassados, segue-se adicionar à lista de leitura o par <mensagem, *timestamp*>. Quando o número de leituras feitas pelo cliente for maior que (N-f)/2, onde N é o número de servidores e f o número máximo de faltas que o sistema tolera, será devolvida a mensagem guardada na lista com o *timestamp* mais elevado.

Relativamente à operação de escrita, as mensagens voltam a ser assinadas e passam a ter um *wts*. Os primeiros passos passam por verificar a assinatura e o *wts* recebido, à semelhança do que acontecia na operação de leitura. Se um servidor responder "ACK", a sua resposta será adicionada à lista. Quando o número de escritas for maior que (N-f)/2, será devolvido "Ok" (o bem passou a estar à venda ou a transferência foi bem sucedida, por exemplo). Esta mensagem será uma resposta correta porque a escrita apenas pode ser efetuada pelo escritor (portanto, escritas que não sejam pelo escritor terão a assinatura incorreta), e o escritor escreve sempre com o *timestamp* mais elevado (pois é correcto), logo o *timestamp* mais elevado é sempre o valor mais recente (e portanto, mais correto).

A figura seguinte ajuda a perceber em que é que se baseou o nosso raciocínio, na hora de implementar este algoritmo :

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bonrr*, *Init* $\rangle$ **do**
$(ts, val, \sigma) := (0, \perp, \perp)$;
$wts := 0$;
acklist $:= [\perp]^N$;
rid $:= 0$;
readlist $:= [\perp]^N$;

upon event $\langle$ *bonrr*, *Write* | v $\rangle$ **do**
$wts := wts + 1$;
acklist $:= [\perp]^N$;
$\sigma :=$ *sign*(*self*, *bonrr*$\|$*self*$\|$WRITE$\|wts\|v$);
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [WRITE, wts, v, σ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [WRITE, ts', v', σ'] $\rangle$ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger $\langle$ *al*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#(acklist) > (N + f)/2$ **then**
acklist $:= [\perp]^N$;
trigger $\langle$ *bonrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
rid := *rid* + 1;
readlist $:= [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [READ, *rid*] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [READ, r] $\rangle$ **do**
trigger $\langle$ *al*, *Send* | p, [VALUE, r, ts, val, σ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [VALUE, r, ts', v', σ'] $\rangle$ **such that** $r =$ *rid* **do**
if *verifysig*(q, *bonrr*$\|w\|$WRITE$\|ts'\|v', \sigma'$) **then**
readlist[q] := (ts', v');
if $\#(readlist) > \frac{N+f}{2}$ **then**
$v :=$ *highestval*(*readlist*);
readlist $:= [\perp]^N$;
trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;

Relativamente ao mecanismo de combate ao *spam* de mensagens por parte dos clientes, criámos o seguinte algoritmo:

- Cada servidor, quando a aplicação fosse lançada, iria gerar n números aleatórios de 4 dígitos. Seria calculo o fatorial desses números, e gerado uma *hash* para esse resultado, que seria guardado internamente por cada servidor. Quando um cliente entrasse na aplicação, cada servidor iria enviar uma *string* que era o resultado de uma das *hashs* guardadas (escolhida de forma aleatória) mais um *salt* (número aleatório de 3 dígitos neste caso).
- Quando um cliente quiser fazer uma transferência, terá de ver a hash que cada servidor lhe enviou e descobrir qual o número aleatório por detrás da mesma, tendo em conta que ainda haveria o *salt* a dificultar o processo. O servidor só aceitará a transferência se receber o número correto por parte do cliente.

Assim, um cliente ao fazer uma transferência irá obrigatoriamente que gastar bastante recursos, dada a alta complexidade deste algoritmo, em termos de tempo e de imensos cálculos. O uso do *salt* irá prevenir que cada cliente tente gerar uma tabela com todos os resultados possíveis, e assim impedir que hajam "apenas" *lookups* de valores em tabelas. Com este aumento de recursos a serem gastos pelo cliente, em cada transferência, irá diminuir o número de solicitações que os servidores podem receber de clientes mal intencionados.

5. Ataques e Ameaças

O uso do tempo de expiração e do número sequencial nas mensagens, referidos na secção 4, faz com que o sistema consiga prevenir ataques de *replays* de mensagens, pois se uma mensagem for enviada mais vezes, depois de já ter sido enviada uma vez, o interveniente em causa irá detetar esse ataque, seja pelo tempo que a mensagem possui já ter expirado, seja pelo número sequencial que a mensagem contém seja menor que o número sequencial que o interveniente possui na altura. Tivemos em conta que o sistema poderá sofrer ataques que impeçam que as mensagens cheguem aos recetores, e logo, caso isto aconteça os números sequenciais irão deixar de coincidir. No entanto, se o número sequencial recebido for maior do que o atual, significa que a mensagem nunca tinha sido recebida (por conseguinte, não é um *replay* ataque), e desta forma poderá ser aceite.

A aplicação tem um número específico de clientes, sendo isto uma medida de defesa contra possíveis ameaças onde o atacante cria propositadamente uma elevada carga de trabalho no sistema (neste caso, poderia ser criar vários clientes diferentes), para levar a uma possível disrupção do servidor, devido a essa excessiva carga no mesmo (levando o servidor ao ponto onde deixaria de ter capacidade de controlar tantos clientes). Caso o *Notary* sofra um *crash*, é feito uma serialização da lista dos bens que existem na aplicação e os respetivos donos, para que assim não haja perda de informação importante. De seguida, quando o *Notary* voltar a correr, irá ler esse ficheiro serializado e recuperar a informação antiga.

Ataques que visem modificar ou corromper as mensagens trocadas no sistema, estão prevenidos usando as assinaturas digitais em cada mensagem.

A implementação do *Byzantine Atomic Register* irá proteger o sistema contra servidores bizantinos (até um certo limite de faltas), portanto mesmo que exista servidores a manipular e enviar mensagens erradas, aos clientes, em resposta às mais diversas operações, os clientes irão conseguir chegar a uma resposta correta.

Finalmente, usando o mecanismo apresentado para combater possível *spam* de mensagens dos clientes para os servidores, diminui drasticamente o impacto que um cliente mal intencionado possa ter no sistema, ao querer fazer muitas transferências num curto espaço de tempo.

6. Limitações do Sistema

O sistema, apesar dos vários métodos e algoritmos desenvolvidos, apresenta algumas limitações. Não foi possível implementar o *Authenticated Echo BroadcastI* que iria resolver problemas em que um cliente é bizantino, está a enviar mensagens válidas mas com conteúdo diferente aos servidores. Logo, este é um caso em que o sistema não se consegue defender. Sabendo o comportamento deste algoritmo, e analisando a sua implementação, onde (aplicando ao nosso contexto), cada servidor ao receber uma mensagem, iria reenviar a mensagem sob a forma de *ECHO* para os restantes servidores, e cada um só iria responder quando recebesse mais do que (N+f)/2 *ECHOS*. Assim, clientes bizantinos a usar este tipo de ataque iriam ser detetados.

A classe referente ao cartão de cidadão encontra-se feita, e presente no código apresentado, mas não se encontra a ser utilizada no nosso sistema. Isto deveu-se ao facto de não termos conseguido testar devidamente a sua implementação, nesta entrega, dado que o nosso leitor de cartões não se encontra a funcionar, e relativamente ao leitor fornecido pelo professor só o conseguíamos aceder em tempo de aula de laboratório, tempo esse que foi escasso para este propósito. Portanto, esta será outra limitação do sistema, dado que o enunciado impunha a sua implementação.