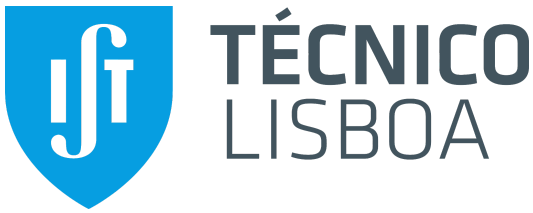

Highly Dependable Systems

Report - Stage 2

2022/2023

Group 35

93747 - Pedro Gomes

96725 - Alexei Calugari

105741 - Francisco António

Canais de Comunicação

Para os canais de comunicação, decidimos optar por Perfect Links. Esta decisão baseia-se no facto de que o algoritmo a implementar, embora necessite que o mecanismo usado para o envio das mensagens seja *Reliable*, lide ele mesmo com os acontecimentos bizantinos. Assim, não considerámos necessário um próximo nível de abstração (Authenticated Perfect Links).

Para atribuir um ID a cada mensagem, e de forma a evitar colisões nas verificações, decidimos que esse identificador seria composto por: <Endereço do servidor>_<Porto de origem>_<Contador de envios para um servidor>

FairLossLink

Simples envio de mensagens UDP onde não há qualquer garantia de ordem ou entrega.

StubbornLink

A ideia do stubborn link é poder transmitir infinitamente uma mensagem, garantindo assim que esta, inevitavelmente, acaba por chegar ao recipiente. De forma a tornar este processo mais eficiente, utilizamos duas estratégias:

1. Utilizar um timeout entre cada retransmissão. Esse timeout cresce exponencialmente a cada tentativa. Evitamos assim situações como ‘Self DoS’.
2. Por cada mensagem chegada ao recipiente, seja esta original ou um duplicado já recebido anteriormente, é enviado de volta um ‘Acknowledge’ (via UDP e portanto sem garantia de chegada). Se esse ACK chegar corretamente à origem da mensagem, a thread responsável por esse envio será interrompida.

PerfectLink

Para garantir que não são entregues mensagens duplicadas, implementámos por fim os Perfect Links. Aqui existe uma estrutura que armazena o ID das mensagens já recebidas, de maneira a poder verificar em que casos se trata de um duplicado.

Primitiva de Broadcast

Neste caso optámos pelo Best-Effort-Broadcast porque, embora seja o mais simples, já estamos a usar canais de comunicação fiáveis e o próprio algoritmo já fornece alguma redundância.

Algoritmo

O cliente envia uma mensagem do tipo “APPEND” para o servidor líder, este, ao receber a mensagem, verifica que é do cliente através do seu tipo e executa o broadcast de mensagens “PRE-PREPARE” para os restantes servidores.
Quando um servidor recebe a mensagem do tipo “PRE-PREPARE”, executa o broadcast de mensagens “PREPARE” com a sua proposta(conteúdo da mensagem).
Quando recebe a mensagem “PREPARE”, se for a primeira vez que recebe a mensagem desse sender(verificação feita através do porto), guarda o sender da mensagem e o conteúdo. Depois, verifica se já recebeu quantidade suficiente de mensagens, ou seja superior a $\frac{2}{3} + 1$ do número total de servidores. Se for o caso, executa o broadcast do “COMMIT” com o conteúdo mais comum das mensagens que recebeu dos “PREPARE”, e de seguida apaga a informação dos senders e o conteúdo das mensagens “PREPARE”.
Quando recebe a mensagem “COMMIT” o processo é idêntico ao “PREPARE”, mas não executa o broadcast, apenas adiciona o bloco à sua chain com o conteúdo mais comum das mensagens “COMMIT” e apaga a informação dos senders e o conteúdo das mensagens “COMMIT”.

Ficou ainda por implementar os timers. Estes seriam de grande importância ao algoritmo, uma vez que seriam o mecanismo que impediria um servidor de ficar “bloqueado” no mesmo pedido de cliente, na espera que um quorum de respostas válidas fosse atingido, o que na presença de servidores Bizantinos, pode nunca acontecer.

Mecanismo de Autenticação

Utilizámos chaves assimétricas para a comunicação, o par de chaves são criadas na criação dos servidores e na criação de cada conta, estas são guardadas nos ficheiros “src/ServerKeys/”, e “src/AccountKeys/” respetivamente. É criado um ficheiro para cada chave e o nome é construído a partir do tipo de chave “pubKey” ou “privKey”, e o id do servidor ou conta, sendo que no caso da chave da conta também tem o id do cliente no seu nome. Desta maneira, sempre que queremos verificar uma assinatura basta obter a chave pública através do ID recebido na mensagem, construindo assim o nome do ficheiro para ler da pasta ServerKeys ou AccountKeys.

De maneira a melhorar a eficiência do mecanismo de autenticação, deveria ter sido implementada uma forma de assinatura baseada apenas em MACs. Esta opção, embora não tão robusta como a utilização de chaves assimétricas, oferece as propriedades necessárias em casos como confirmações de Commit (Decide), e em algumas comunicações entre servidores (as que não requerem um Justify).

Modos de Leitura

Weakly consistent read:

Na implementação do weakly read adicionamos um contador em todos os servidores para que de x em x blocos adicionados à blockchain, todos os servidores criam um snapshot dos estados das contas (TES.java) e adicionam a sua assinatura para aquele estado. O contador é incrementado sempre que se adiciona um bloco, e quando chega ao

número definido, o servidor cria o snapshot que vai conter os saldos das contas todas, adicionando a sua assinatura ao snapshot e o seu id. De seguida é feito o broadcast “PREPARE” desse snapshot. Ao receber esse “PREPARE”, é verificado que foi recebido um snapshot e é tratado como um bloco. Quando é atingido o quorum, o servidor cria uma nova instância do seu snapshot e percorre os snapshots recebidos, se o estado das contas for igual ao quorum value(valor mais comum) então atualiza esse estado no seu novo snapshot e adiciona a assinatura à lista de assinaturas juntamente com o server id que criou essa assinatura. No fim o seu novo snapshot vai ter o estado das contas, e a lista de assinaturas dos servidores que mandaram o snapshot no mesmo estado juntamente com os seus ids. Quando o cliente recebe essa resposta apenas verifica as assinaturas e se tem o número suficiente(F + 1), assim tem a certeza que o mesmo estado de contas foi assinado por F + 1 servidores.

Strongly consistent read

Para o strongly, a lógica foi a seguinte:

1. Perguntar a todos os servidores, qual o valor que tem armazenado na sua instância do TES, para uma conta específica;
2. Reunir todas as respostas e tentar obter um Quorum de respostas válidas;
3. Devolver esse valor ao cliente

Blocos e transações

O que são transações?

Definimos que uma transação é qualquer tipo de operação que resulta num Update do estado do Token Exchange System (TES), ou seja, operações de Create Account e Transfer.
O Check Balance, por ser apenas uma Query, não pertence às transações.

Transações

Quando uma transação chega de um cliente ao servidor líder, este adiciona-a a um bloco.

Validação de Transações

A validação das transações é feita individualmente, não tendo em consideração qualquer outra transação anterior ao nível desse bloco em específico, ou seja, não é possível assumir que uma transação é válida tendo como base uma transação do mesmo bloco. Estas validações baseiam-se apenas no estado do TES, que é o já concordado por todos os membros.

Blocos

Assim que o bloco atual atinge o número específico de transações, procedemos à validação individual de cada uma dessas transações.
Se o resultado dessa validação for positivo, então o bloco está pronto para ser levado ao *consensus*. Posteriormente, se tudo correr como esperado, esse bloco será aceito e adicionado à própria blockchain.

Mecanismo de Segurança

Toda a comunicação é feita através de assinaturas com o uso de chaves assimétricas. Embora este mecanismo pudesse ser substituído por MACs em algumas situações (como já referido acima), no TES é a única solução, dado que, um dos requisitos das suas operações é a garantia de non-repudiation, algo que as chaves assimétricas garantem, assumindo que a PrivateKey não tenha sido comprometida.

Decidimos ainda, nas estruturas para as quais geramos assinaturas, adicionar um timestamp. Desta forma, conseguimos garantir sempre um valor diferente (funcionando à semelhança de um nonce) e prevenir Replay-Attacks.