

# Trabalho Final - Fundamentos de Redes de Computadores

Thiago Zilberknop, Luigi Salvatore, Leonardo Rosa

O trabalho final de Fundamentos de Redes de Computadores demanda a implementação de uma aplicação que simula uma rede local em anel. A rede é pré configurada através de arquivos de configuração que contém informações referentes aos nós da rede, e também define a topologia geral do projeto. Os nós da rede devem ser capazes de gerenciar o token que autoriza a comunicação de mensagens entre os dispositivos da rede. O nodo deve saber receber os pacotes com os dados da mensagem e saber gerenciar e processar as informações presentes no pacote.

## Implementação - Variáveis de um Nó na Rede:

O nó na rede é definido na classe Node; cada nó na rede pertence a essa classe. As variáveis dentro da classe definem as diferentes informações da topologia do nó:

Variáveis contidas dentro do arquivo de configuração:

*self.dest\_ip* → Endereço IP do nodo destino, localizado a direita do nodo atual

*self.dest\_port* → Porta do nodo destino

*self.name* → Nome do nodo atual

*self.token\_lifetime* → Tempo de token para o nodo atual

*self.has\_token* → Booleano que define se o token está presente ou não

Variáveis de conexão na rede:

*self.listen\_port* → Porta do nodo atual utilizada para receber mensagens (hardcoded 11000)

*self.msgs* → Lista de mensagens possíveis para envio do nodo

*self.msg\_lock* → Lock para gerenciamento de mensagens

*self.socket* → Socket para conexões

Variáveis de Gerenciamento de Token:

*self.is\_token\_manager* → Booleano que define se o nodo é o atual gerenciador do token

*self.token\_time* → Timer que calcula o tempo que o token ficou no nodo atual

*self.last\_token\_time* → Armazena o tempo da última vez que o nodo atual recebeu o token

*self.token\_lifetime* → Mesmo valor do token\_lifetime do arquivo, convertido para inteiro

*self.token\_timeout* → Tempo de timeout do token em segundos

*self.token\_timemin* → Tempo mínimo que deve ser passado até que o token possa ser enviado

Variáveis de injeção de erro na rede:

*self.corrupt\_next\_message* → Booleano que informa se a próxima mensagem deve ser corrompida

*self.corrupt\_field* → Define qual parte da mensagem deve ser corrompida

Variáveis utilizadas na GUI:

*self.log\_queue* → Queue que contém informações de log

### **Implementação - Threads:**

O Nodo deve exercer múltiplas funções simultâneas para ser utilizado corretamente. Excluindo o gerenciamento e atribuição dos dados dos pacotes para as suas respectivas variáveis dentro da sua classe, o nodo também deve estar sempre escutando por potenciais mensagens sendo recebidas, deve sempre estar preparado para gerenciar o token e também deve sempre estar preparado para disponibilizar e mostrar a GUI caso for pedida pelo usuário. Para possibilitar a execução de múltiplas tarefas simultâneas, a utilização de threads é necessária na topologia. As threads utilizadas são o seguinte:

*listen\_loop* thread → Thread que está sempre escutando para potenciais mensagens recebidas

*token\_handler* thread → Thread que está sempre preparada para gerenciar o token

*start\_gui* thread → Thread que gerencia as variáveis da GUI do nodo

#### *listen\_loop* thread:

A lógica por trás da função *listen\_loop()* é simples: esperar informações de outros dispositivos. A função será executada infinitamente através de sua condição *while True*, que garante que o programa irá ficar preso dentro do while até ser desligado. Dentro do while, o programa espera receber dados pelo socket através da porta definida na variável *listen\_port*, a porta definida anteriormente que representa o local onde as mensagens serão recebidas. Após o recebimento da mensagem, ela é extraída e dividida em seus respectivos componentes. A variável *raw\_msg* armazena os dados da mensagem após serem convertidas para o formato utf-8. O programa então confere se a mensagem recebida é um string “9000”, demonstrando que um token foi recebido. Se a mensagem for um token, a função *self.handle\_token()* é chamada. Se a mensagem não for um token, ela consequentemente é um pacote de dados, que é gerenciado na função *self.handle\_message()*.

#### *token\_handler* thread:

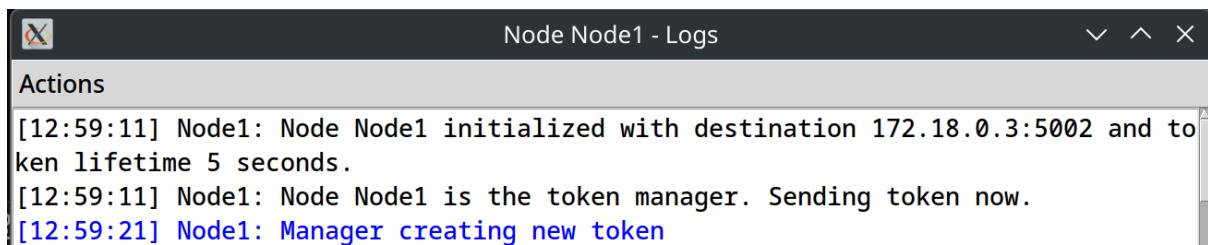
A thread *token\_handler* está sempre verificando se o nodo atual está com o token ou não. Se o nodo não está com o token presente, a thread continua esperando até o token ser recebido. Para garantir que não haverá problemas futuros no gerenciamento do token, somente o *token\_manager* pode executar operações referentes ao token. O *token\_manager* é o nodo no qual tem seu booleano do token setado como *true* no arquivo de configuração. Se o nodo atual é o *token\_manager*, a função *token\_handler* verifica se o token está fora do tempo de timeout. Se sim, um novo token é criado e distribuído pela rede. Essa thread foi criada para garantir que a topologia sempre terá um token; quando um token é deletado/removido da rede, a thread detecta esse comportamento e cria um novo token nessa situação.

*start\_gui* thread:

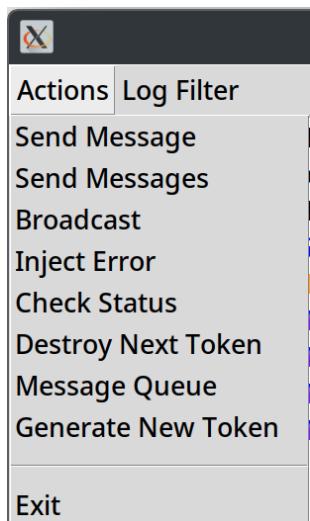
Diferente das outras, a thread *start\_gui* não está presa em um loop infinito definido dentro da thread. Essa thread é utilizada para inicializar os componentes da GUI. Variáveis como o nodo, informações sobre o nodo e menus de comando são todos inicializados e armazenados dentro de logs presentes na GUI. Após sua inicalização, o programa executa em sua própria main, através do *root.mainloop()*, que garante que a GUI está sempre presente até que a sua aba é fechada.

### Implementação - GUI:

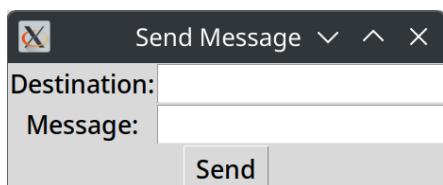
Mesmo não sendo um requisito do trabalho, nós acreditamos que a implementação de uma GUI seria muito mais benéfico do que ruim para a nossa implementação. Acreditamos que uma GUI facilita a compreensão de cada função presente no nodo, devido à sua interface simples e fácil de entender. A nossa GUI contém duas partes importantes: um ‘terminal’ que disponibiliza todas as informações relevantes da rede, e uma aba que disponibiliza ao usuário conseguir interagir com a topologia de uma forma mais interativa. O nosso terminal descreve informações como recebimento do token, mensagens enviadas e ACKs recebidos das mensagens, o conteúdo da mensagem e outras informações semelhantes.



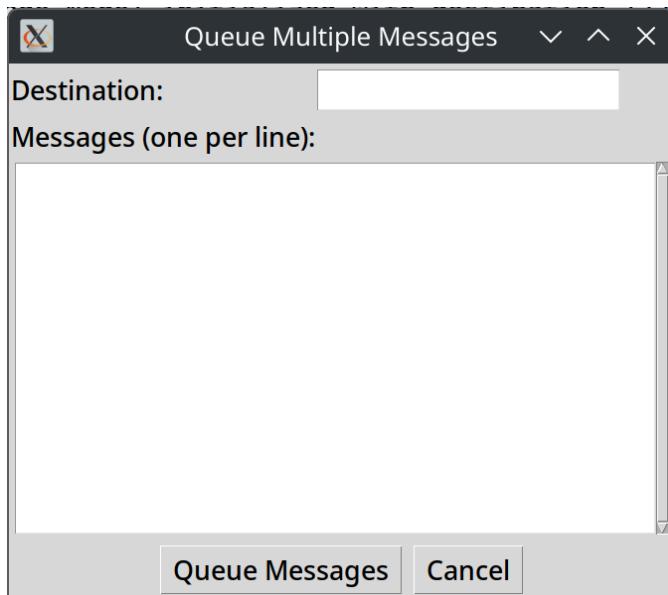
Clicando na aba *Actions*, temos as seguintes funções:



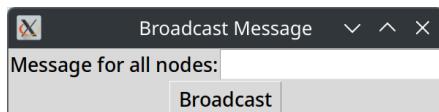
*Send Message* → Envia uma mensagem para um nodo presente na rede



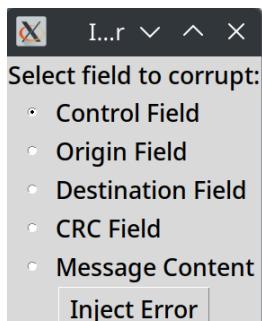
*Send Messages* → Envia uma lista de mensagens para um nodo presente na rede



*Broadcast* → Envia uma mensagem para todos os nodos presentes na rede



*Inject Error* → Acrescente um erro dentro da rede para avaliar a redundância da topologia



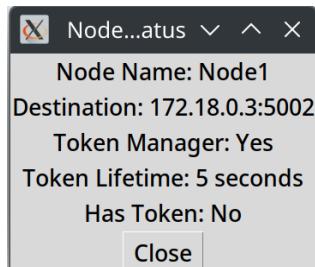
*Generate New Token* → Permite que qualquer nodo consiga criar um novo token na rede

[10:48:35] Node1: Sending new token!

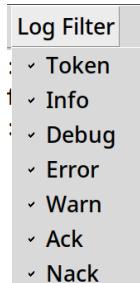
*Destroy Next Token* → Deleta o próximo token recebido pelo nodo

[10:48:59] Node1: Deleting next token!

*Check Status* → Mostra informações sobre o nodo



Clicando na aba *Log Filters* podemos filtrar as possíveis mensagens de log que aparecem no terminal:



#### Envio de Mensagens:

A função *send\_message\_dialog* permite que o usuário envie uma mensagem diretamente para outro nó. Ao clicar na opção “Send Message”, uma nova janela se abre com campos para inserir o nome do destinatário e o conteúdo da mensagem. Após o envio, a mensagem é enfileirada internamente por meio da função *queue\_msg*, que também calcula e armazena o valor de CRC da mensagem.

Outra funcionalidade disponível na GUI é o envio de mensagens de broadcast, descrita pela função *broadcast\_dialog*. Ao utilizar essa opção, o usuário pode enviar uma única mensagem para todos os nós da rede. Internamente, isso é feito ao definir o destinatário da mensagem como "TODOS", que é interpretado pelo código como uma instrução para processar a mensagem independente de qual for o nó.

#### Injeção de Erro:

Para ver se os mecanismos de erro estão funcionando corretamente, a GUI permite injetar erros nas mensagens. Ao selecionar a opção “Inject Error”, o usuário pode escolher qual campo da próxima mensagem será corrompido: controle, origem, destino, CRC ou conteúdo. O próximo envio programado será alterado pela função *apply\_corruption*, simulando um erro proposital. Todo esse gerenciamento, incluindo as opções escolhidas pelo usuário na GUI são definidas na função *inject\_error\_dialog*.

#### Status:

Por fim, a GUI oferece uma opção chamada “Check Status”, que abre uma janela informando os dados atuais do nó: nome, IP e porta de destino, se o nó é o gerenciador do token, se o token está presente, e o tempo de vida do token. Essa funcionalidade, implementada pela função *status\_dialog*, permite que o usuário monitore facilmente o estado do nó local em tempo real. Todos os valores necessários para imprimir o status estão presentes na classe Nodo, e os mesmos já contém os seus valores corretos na inicialização. O programa então obtém esses valores e imprime eles para visualização do usuário.

#### Logs:

Decidimos que utilizar logs para informar o usuário das atividades executadas por cada nodo seria algo útil não só para interpretar o que está acontecendo dentro da rede mas

também para debugar eventuais problemas que podem surgir durante a execução. A função `update_logs` é responsável por manter o terminal da interface gráfica sempre atualizado com as mensagens de log que descrevem os eventos ocorridos no nó. Essa função atua como um loop de atualização contínuo, sendo executada de forma periódica através do método `root.after(100, update_logs)`. As mensagens de log são armazenadas dentro de um queue, onde são eventualmente imprimidas para visualização do usuário.

## Implementação - Leitura do Arquivo de Configuração

Sabendo que todos os dados relevantes do nodo ficam localizados dentro de um arquivo .conf, temos que desenvolver uma forma de adquirir as informações presentes dentro do arquivo. A melhor forma de fazer isso foi fazer a leitura do arquivo ser a primeira tarefa da inicialização. O usuário deve informar o arquivo de configuração desejado como parâmetro na inicialização do nodo; desta forma, fica claro qual arquivo de configuração corresponde para qual nodo da rede. Quando o nodo é inicializado, o arquivo é lido linha por linha até o seu fim. Sabendo que o arquivo contém um formato fixo, decidimos armazenar as informações nas variáveis de forma estática. A variável `lines`, que recebe o conteúdo do arquivo, é dividida em quatro partes, cada uma delas correspondendo a uma linha do arquivo. Essa mesma variável é então dividida da seguinte maneira:

*Strip 1* → Contém o IP destino e a porta; Inserida na variável `dest`.

*Strip 2* → Contém o apelido da máquina atual; Inserida na variável `self.name`

*Strip 3* → Contém o tempo do token; Inserida na variável `self.token_lifetime`

*Strip 4* → Contém um booleano do token; Inserida na variável `token_status`

Um ponto importante de abordar é que mesmo sendo um booleano, a variável `token` do arquivo é adquirida como uma string. Para resolver essa diferença, temos que incluir uma condição para enviar o conteúdo da string em seu formato adequado para a variável do nodo. Se a string do arquivo é a palavra ‘true’, a variável `self.has_token` é verdadeira. Se a string do arquivo é a palavra ‘false’, a variável `self.has_token` é falsa.

## Implementação - Gerenciamento de Mensagens:

### Função handle\_message:

Quando um nodo recebe uma mensagem, ele deve saber lidar com suas informações da forma correta. A mensagem consiste de cinco parâmetros de dados que devem ser extraídos e processados pelo nodo: controle do erro, apelido da origem, apelido do destino, CRC e a mensagem em formato string. A lógica por trás da separação de cada dado, e na atribuição das informações para as suas designadas variáveis é muito semelhante à estratégia utilizada na leitura do arquivo. A ideia é que a mensagem será recebida por inteira como parâmetro e dividida em suas designadas partes através da função `split`. É importante ressaltar que a primeira informação da mensagem, o controle de erro, é dividida em duas partes: o tipo da mensagem e uma string de controle. Para garantir que todas as informações são extraídas

corretamente, um split é executado antes da separação dos demais blocos, específico para esse dado. Após essa separação, as variáveis que receberão as informações são as seguintes:

*msg\_type* → Recebe a sequência numérica no início da mensagem

*control* → Recebe a string de controle da mensagem (“naoexiste”, “ACK”, ou “NACK”)

*origin* → Recebe o nome da máquina de origem da mensagem

*dest* → Recebe o nome da máquina de destino da mensagem

*crc* → Recebe o valor de CRC da mensagem

*msg\_content* → Recebe o conteúdo da mensagem

Sabendo que todas as mensagens obrigatoriamente contém o seu dígito inicial do controle de erro como “7777”, podemos aplicar a lógica de gerenciamento de mensagem somente para pacotes com inicial “7777”, descartando e mostrando uma mensagem de erro caso o pacote não contenha esse dígito.

Identificando a mensagem, algumas outras verificações devem ser feitas para processar todas as informações corretamente. Se a mensagem é para o nodo atual, ou seja, a variável *dest* é igual a variável *self.name*, a função *process\_message* é chamada para gerenciar o conteúdo da mensagem. Existe a possibilidade de acontecer um *loopback* no envio das mensagens, ou seja, uma mensagem enviada é recebida pelo mesmo nodo; essa verificação é feita através da comparação do parâmetro *origem* e do nome do nodo atual. Se for o caso, algumas coisas devem ser feitas:

Se a mensagem recebida conter um “NACK” em sua variável de controle, sabemos que ela é uma mensagem anterior que não foi enviada corretamente. Neste caso, verificamos se a mensagem já foi reenviada. Se sim, a mensagem é descartada. Se não, a mensagem é inserida na fila de mensagens para reenvio na próxima passagem do token.

Se a mensagem recebida conter um “ACK”, sabemos que a mensagem enviada foi recebida corretamente. Neste caso, uma mensagem é incluída no log confirmando o seu recebimento.

Se a mensagem recebida conter um “naoexiste”, sabemos que a mensagem não foi enviada por ausência de um destino. Neste caso, um warning é incluído no log.

A última possibilidade no recebimento da mensagem é o recebimento de uma mensagem que não foi enviada pelo nodo atual e não é destinada para o nodo atual. Neste caso, a mensagem é *forwarded*.

#### Função process\_message:

Quando o destino da mensagem é o nodo atual, identificado na função *handle\_message*, a função *process\_message* é chamada. Essa função recebe os parâmetros *control*, *origin*, *crc* e *message\_content*, todos extraídos pela função anterior. O principal objetivo da função é calcular o CRC da mensagem e conferir com o CRC presente dentro da mensagem. Se o CRC calculado é igual ao CRC enviado por parâmetro, um “ACK” é

atribuído na variável de controle. Se o CRC não for igual ao CRC enviado por parâmetro, um “NACK” é atribuído na variável de controle. A mensagem então é reconstruída e enviada para o nodo de origem.

Para calcular o valor de CRC, a função *crc32*, presente na biblioteca *zlib* da linguagem Python, é utilizada. Essa função computa um checksum CRC32 em formato de *unsigned int* de 32 bits para qualquer string fornecida à função. A vantagem de utilizar essa função é que o mesmo valor de CRC será gerado para a mesma string, independente de possíveis configurações de ambiente. Quando há uma falha na mensagem enviada, o valor de CRC irá ser alterado no seu cálculo, ativando a condição de atribuição do “NACK” na mensagem. É importante ressaltar que as funções *handle\_message* e *process\_message* estão sempre sendo executadas através de sua thread paralela.

### **Implementação - Gerenciamento de Token:**

O gerenciamento de token é feito através da função *handle\_token*, que é chamada quando a função *listen\_loop* identifica que a mensagem recebida é um token. A primeira coisa verificada na função é se a variável *del\_next\_token* é verdadeira. Essa variável define se o token recebido pelo nodo deve ser deletado, e a sua condição de ativação depende do input do usuário na GUI. Sabendo que o token não será deletado, iniciamos o timer que calcula a quantidade de tempo que o token está no nodo. Após sua inicialização, verificamos se o nodo atual é o *token manager*. Se sim, e se o token foi recebido muito antes do previsto, o token é ignorado. Se não, sabemos que as funções padrões do token devem ser exercidas.

Utilizamos um *lock* nas mensagens quando estamos gerenciando o token por causa de possíveis conflitos nas mensagens quando são gerenciados pela thread rodando em paralelo. Desta forma, sabemos que qualquer erro nas mensagens não é devido a conflitos nas threads. Primeiro setamos as variáveis de ‘controle’ do nodo referente ao token. Definimos que o token está presente no nodo atual e atualizamos o seu tempo de token no nodo. Se há mensagens para serem enviadas na sua lista de mensagens, a primeira mensagem é retirada da fila através da função *pop(0)* e a mesma é enviada para o próximo nodo. Caso não haja mais mensagens para serem enviadas, o token é enviado para o próximo nodo.

### **Implementação - Erros:**

Incluso na função *handle\_token* tem uma lógica para corromper mensagens. Essa *feature* foi acrescentada por nós na nossa GUI, com a intenção de verificar se o mecanismo de detecção de erros foi implementado corretamente. O corrompimento é feito de uma forma simples: o usuário irá definir qual campo da mensagem será alterado através da GUI. Dependendo do que ele escolher, o conteúdo da mensagem será alterado para algo impossível de ser reconhecido corretamente pelo nodo. Por exemplo, se eu quero verificar se o cálculo do CRC está funcionando corretamente, eu declaro na GUI que a próxima mensagem a ser enviada será corrompida no “CRC Field”. O nodo vai identificar na função *handle\_token* que uma corrupção deve ser feita na mensagem, e portanto, envia essa mensagem para a função *apply\_corruption*. Essa função detecta qual campo da mensagem deve ser corrompido e aplica seu corrompimento de acordo. Já que queremos que o CRC seja corrompido no nosso

exemplo, a função irá alterar o campo de CRC e retornar a mensagem para o *handle\_token*. A mensagem, portanto, será enviada de forma incorreta, disparando o nosso detector de erros.

### Execução:

Para inicializar um nodo na rede, é necessário utilizar o comando:

*python3 main.py <arquivo de configuração> <porta de listen do nodo>*

No exemplo a seguir, estamos inicializando dois nodos na rede: o Nodo1 e o Nodo2. Os seus respectivos arquivos de configuração contém as seguintes informações:

node1.conf:

172.18.0.3:5002

Node1

5

true

5001

node2.conf:

172.18.0.2:5001

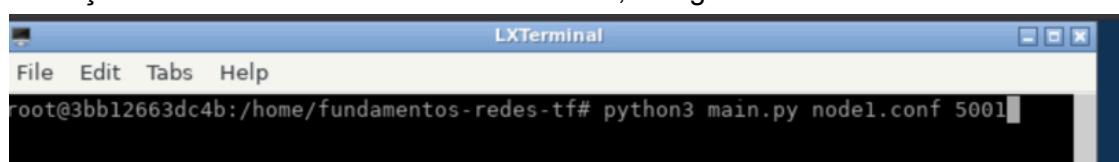
Node2

5

false

5002

Observa-se que há um parâmetro adicional na última linha de cada arquivo. Esse parâmetro não é lido pelo nodo, e representa o valor da *listen\_port* de cada nodo. Ele foi adicionado no arquivo de configuração como uma conveniência durante os testes de execução. Para inicializar os nodos no terminal, o seguinte deve ser feito:

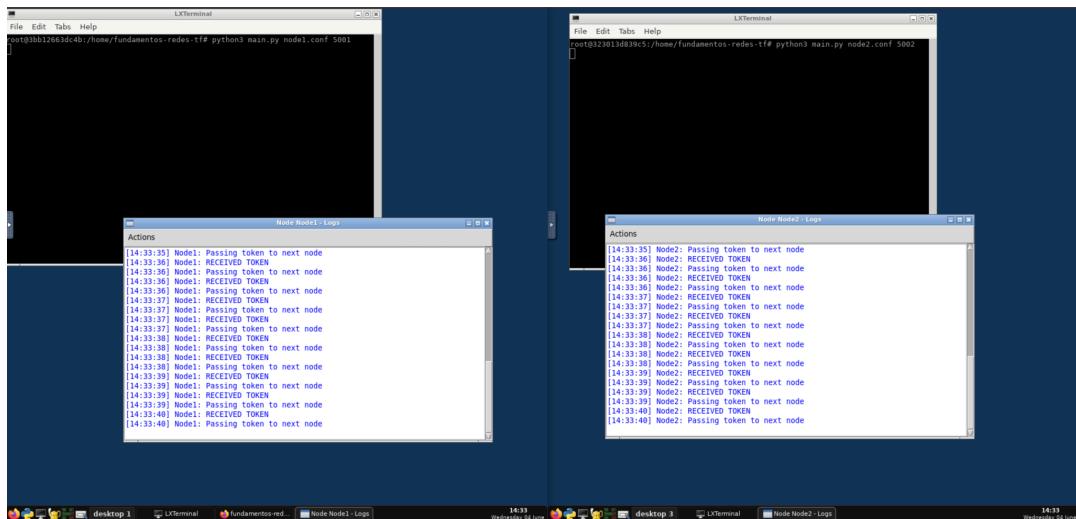


```
LXTerminal
File Edit Tabs Help
root@3bb12663dc4b:/home/fundamentos-redes-tf# python3 main.py node1.conf 5001
```

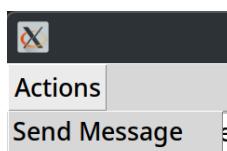


```
LXTerminal
File Edit Tabs Help
root@323013d839c5:/home/fundamentos-redes-tf# python3 main.py node2.conf 5002
```

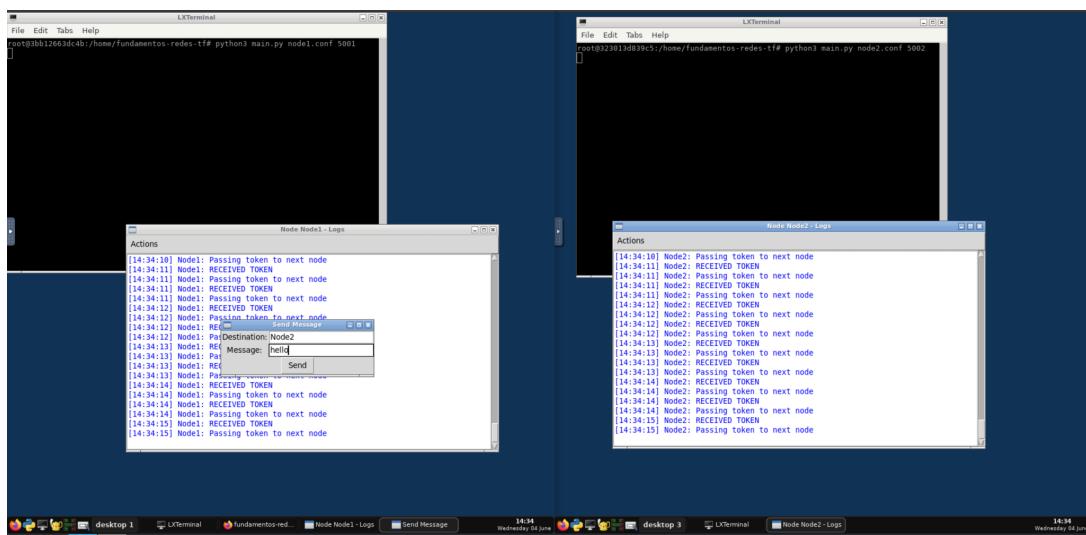
Quando executado, a GUI de cada nodo será aberta:



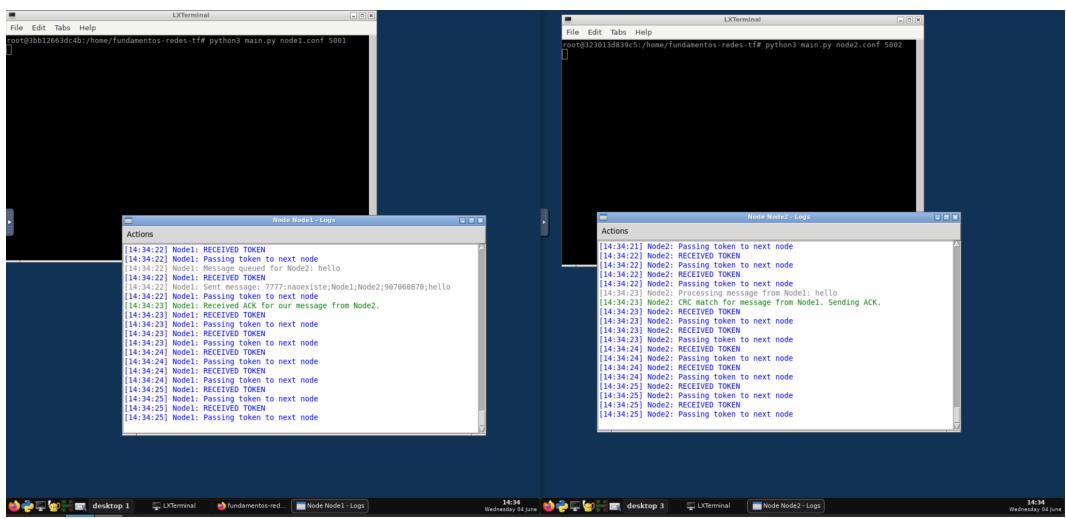
Se o usuário quiser enviar uma mensagem, ele clica na aba “Actions” e depois em “Send Message”.



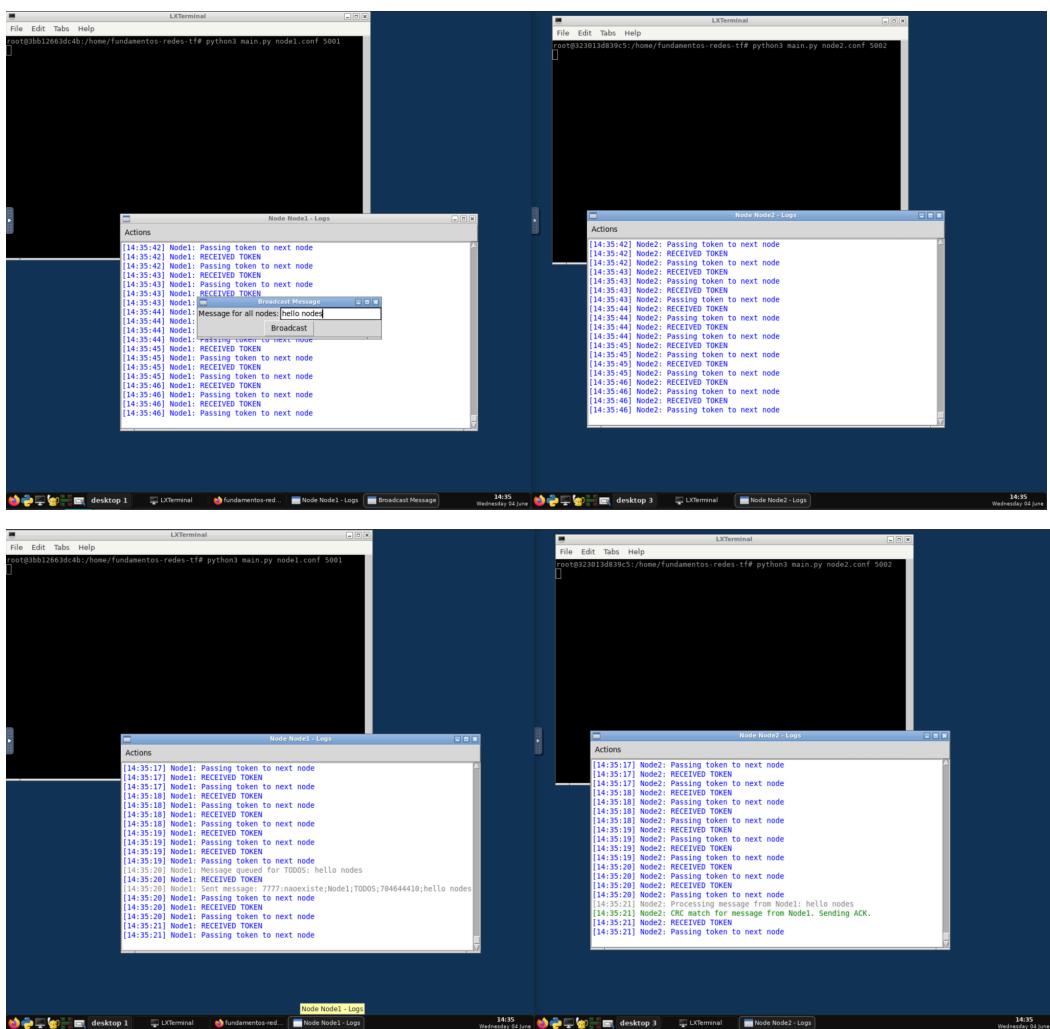
O menu seguinte será aberto:



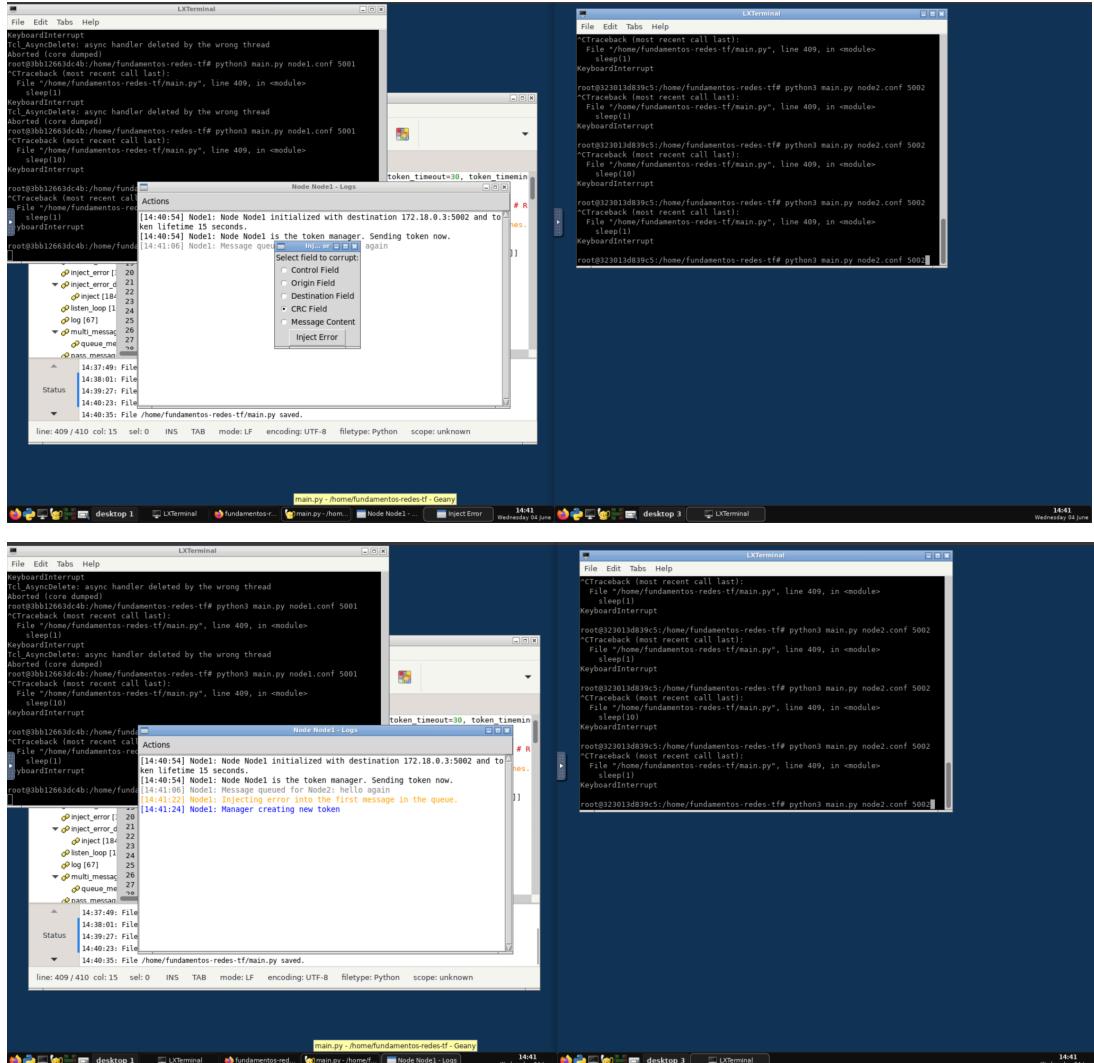
O usuário pode então digitar a sua mensagem e o destinatário, e quando o botão “Send” é clicado, a mensagem é enviada para a fila de mensagens. No caso da topologia atual, como somente há dois nodos, a mensagem será enviada diretamente para o nodo de destino. O comportamento esperado do nodo destino é receber a mensagem, calcular o seu CRC e devolver um ACK confirmindo que a mensagem enviada foi recebida e interpretada com sucesso. Esse comportamento é o que observamos a seguir:



O mesmo princípio acontece quando se quer mandar uma mensagem para todos os nós:



Para inserir um erro em uma mensagem da rede, a mensagem deve ser criada e inserida dentro do queue de mensagens do nodo. Depois, o menu de “Inject Error” deve ser aberto:



Nesse exemplo, um erro de CRC foi injetado na primeira mensagem da fila de mensagens do nodo. O comportamento esperado dessa situação é que o Nodo2 vai receber a mensagem e detectar o erro de CRC, inserindo um “NACK” na mensagem e a enviando para o Nodo1:

Por fim, o menu de “Check Status” mostra informações relevantes de cada nodo para o usuário: