EGAP - Electrical Grid Alert Protocol

Artur Fonseca Costa

1 Introdução

O protocolo EGAP (Electrical Grid Alert Protocol) é uma resposta às falhas críticas em redes elétricas, geralmente causadas por uma reação em cadeia, muito comuns no mundo todo. O EGA será executado em redes de sensores sem fio (WSNs), para que seja possível detectar falhas em certas partes de uma rede elétrica e prevenir a reação em cadeia que possa gerar uma falha crítica.

O enunciado original propõe diversos tipos de mensagens e maneiras de lidar com as mensagens de fluxo de controle, mas diversas instruções eram contraditórias, ambíguas, ou simplesmente falhas. Portanto, o protocolo EGAP apresentado aqui conserta os problemas de funcionamento presentes no enuciado, sem alterar as especificações de entrada e saída.

A Seção 2 mostra as mensagens utilizadas no protocolo. A Seção 3 detalha melhor o funcionamento do servidor, explicando os comandos que ele recebe via teclado. A Seção 4 detalha melhor o funcionamento do cliente, explicando os comandos que ele recebe via teclado. O resto desta seção detalha alguns dos problemas encontrados no enunciado original.

Fora uma quantidade considerável de informações desnecessárias (definição de pane elétrica, detalhamento da detecção de erros por um sensor, etc.), existem também problemas de coerência no enunciado original. Existem instruções conflitantes sobre como definir o ID do servidor e do cliente, assim como a localização e o status do cliente. Não apenas isso, mas algumas mensagens carregam informações redundantes, considerando que o receptor possui essa informação. Também não há uma coerência sobre quais informações o servidor deve imprimir na tela (e.g. imprime *Sensor limit exceeded*, mas não *Peer limit exceeded*). O cliente também é chamado de sensor, equipamento e usuário, às vezes trocando nomes de variáveis que deveriam ser únicas. Por fim, o cliente e o servidor podem imprimir mensagens em inglês e português, sendo também incoerente.

2 Mensagens

Todas as mensagens possuem um header de 1 byte e um payload de tamanho variável. O enunciado define um tamanho máximo de 500 bytes por mensagem, mas a maior mensagem com as especificações do enunciado é de 151 bytes. Em todo caso, foi definido um ID de 10 dígitos (10 bytes) e um máximo de 15 clientes simultâneos. Segue abaixo uma lista das mensagens, contendo o seus códigos e possíveis descrições adicionais.

- `OK` (código 0): representa uma operação bem-sucedida. O payload é o código da mensagem de sucesso, sendo apenas 1 byte. Os possíveis códigos são `SUCCESSFUL_DISCONNECT` (01), `SUCCESSFUL_CREATE` (02), `SENSOR_STATUS_OK` (03), e `SUCCESSFUL_CONNECT` (04).
- `REQ_CONNPEER` (código 20): representa uma requisição de conexão com o servidor que está ouvindo na porta P2P. O payload é o ID do servidor candidato, sendo então 10 bytes. Este ID já foi classificado como único pelo servidor ouvinte, então esta mensagem não retorna erros de protocolo.
- `RES_CONNPEER` (código 21): representa a resposta do servidor ouvinte ao servidor que quer se conectar na porta P2P. O payload é o ID do servidor ouvinte, sendo então 10 bytes.
- `REQ_DISCPEER` (código 22): representa a requisição de uma conexão P2P, que pode ser enviada pelo servidor ouvinte ou não. O payload é o ID do servidor requisitando a desconexão, sendo então 10 bytes.
- `REQ_CONNSEN` (código 23): representa uma requisição de conexão cliente-servidor. O payload é o ID do cliente candidato, sendo então 10 bytes. Este ID já foi classificado como único pelo servidor, então esta mensagem não retorna erros de protocolo.
- `RES_CONNPEER` (código 24): representa a resposta de uma conexão cliente-servidor bem-sucedida. Esta mensagem não possui payload.
- `REQ_DISCPEER` (código 25): representa a requisição de uma conexão cliente-servidor, que só pode ser enviada pelo cliente. O payload é o ID do cliente requisitando a desconexão, sendo então 10 bytes.

- `REQ_CHECKALERT` (código 36): representa um pedido do servidor de status ao servidor de localização, para que retorne a localização de um cliente cujo status é de alerta. Como ambos os servidores registram os mesmos IDs para os clientes, não é necessário passar o ID, apenas o índice do cliente dentro das estruturas de dados do servidor. Logo, o payload é de 1 byte.
- `RES_CHECKALERT` (código 37): representa a resposta do pedido de localização feita por um servidor. O payload desta mensagem também é de 1 byte, contendo a localização do cliente.
- `REQ_SENSLOC` (código 38): representa o pedido de um cliente pela localização de um outro cliente. O payload é o ID do cliente cuja localização se deseja obter, sendo então 10 bytes.
- `RES_SENSLOC` (código 39): representa a resposta do pedido de localização feita por um cliente. O payload desta mensagem é a localização do cliente requisitado, sendo apenas 1 byte.
- `REQ_SENSSTATUS` (código 40): representa o pedido de um cliente pelo status de um outro cliente. O payload é o ID do cliente cujo status se deseja obter, sendo então 10 bytes.
- `RES_SENSSTATUS` (código 41): representa a resposta do pedido de status feita por um cliente. O payload desta mensagem é a localização do cliente requisitado, sendo apenas 1 byte.
- `REQ_LOCLIST` (código 42): representa o pedido de um cliente pelos IDs de todos os clientes numa dada localização. O payload são os caracteres da localização, sendo então 2 bytes.
- `RES_LOCLIST` (código 43): representa a resposta do pedido da lista de clientes numa dada localização. O payload desta mensagem é a lista de IDs. Se existem n clientes na localização requisitada, o payload é de $10n$ bytes.
- `REQ_VALIDATEID` (código 50): representa um pedido de validação de ID, que pode ser enviado tanto por um servidor quanto por um cliente. O payload é o ID que se deseja validar, sendo então 10 bytes.
- `RES_VALIDATEID` (código 51): representa a resposta do pedido de validação de ID, que sempre é enviada por um servidor. O payload desta mensagem é `UNIQUE` ou `NOT_UNIQUE`, sendo então apenas 1 byte.
- `REQ_SERVERIDENTITY` (código 60): representa um pedido de identificação feito por um servidor, que pode ser feito tanto para outro servidor quanto para um cliente. Esta mensagem não possui payload.
- `RES_SERVERIDENTITY` (código 61): representa a resposta do pedido de identificação feito por um servidor. O payload desta mensagem é `IDENTITY_STATUS`, `IDENTITY_LOCATION`, ou `IDENTITY_NONE`, se não é possível determinar.
- `ERROR` (código 255): representa uma operação que falhou. O payload é o código da mensagem de falha, sendo apenas 1 byte. Os possíveis códigos são `PEER_LIMIT_EXCEEDED` (01), `PEER_NOT_FOUND` (02), `PEER_LIMIT_EXCEEDED` (09), e `SENSOR_NOT_FOUND` (10), e `LOCATION_NOT_FOUND` (11).

3 Servidor

Cada servidor possui um socket para ouvir novos clientes, e potencialmente um para ouvir outros servidores (*peers*). Ao ser inicializado, pelo comando

`./server <IP> <porta_P2P> <porta_clientes>`,

ele tenta se conectar a `IP:porta_P2P`. Se essa conexão falha, ele é o primeiro servidor a ser executado e passa a ouvir nessa porta. Se a conexão não falha, ele se conecta ao servidor que já está ouvindo. Se o número máximo de conexões P2P foi atingido, o servidor ouvinte envia a mensagem `ERROR(01)` e encerra a tentativa de conexão. Caso contrário, o servidor ouvinte envia a mensagem `OK(04)`, informando que o servidor candidato pode proceder com a conexão.

O servidor possui um ID único. Para garantir que esse ID seja único, o servidor que ouve na porta P2P define o seu próprio ID. Outros servidores, após se conectarem na porta P2P, enviam IDs ao servidor ouvinte com `REQ_VALIDATEID` para que ele possa testar a unicidade. O servidor ouvinte responde se o ID é único ou não com `RES_VALIDATEID`. Esse processo se repete até que o servidor candidato receba a resposta de que o ID é único, enviando então `REQ_CONNPEER` para iniciar a conexão P2P. Diferente do enunciado original, onde os IDs eram definidos por conexão, o ID definido por um servidor se mantém até o fim da sua execução.

O servidor também possui um identificador (`IDENTITY_STATUS` ou `IDENTITY_LOCATION`), que ele usa para decidir quais informações guardar e também detectar erros. A identificação é feita por meio do cliente, enviando uma mensagem `REQ_SERVERIDENTITY`. O cliente, que recebe as portas dos dois servidores separadamente, sabe exatamente quem lhe enviou a requisição, respondendo com `RES_SERVERIDENTITY`. Caso os clientes se desconectem e um dos servidores seja trocado, essa informação é retida pelo servidor que foi mantido, não sendo necessário depender dos clientes para identificação. Neste caso, o servidor candidato envia uma mensagem `REQ_SERVERIDENTITY` ao servidor ouvinte.

A partir deste ponto, um servidor só pode receber do teclado os comandos `kill` e `close connection`. Se existe uma conexão P2P ativa, ambos os comandos requisitam a desconexão e encerram a execução. Se não existe uma conexão ativa, no entanto, o comando `kill` encerra a execução, enquanto o comando `close connection` não. Se a conexão P2P for desfeita enquanto há clientes conectados nas portas, todas as conexões são desfeitas e o sistema inteiro é desligado. O enunciado original não dá nenhum indício de que essa situação deva ocorrer em uso normal, então essa resiliência não foi implementada no EGAP.

4 Cliente

Cada cliente também possui um ID único, que é definido de forma similar ao ID do servidor. Por meio de uma mensagem `REQ_VALIDATEID`, o cliente pode gerar um ID aleatório e perguntar aos servidores se ele é único. Os servidores guardam os mesmos IDs de clientes, então o teste passa em ambos ou em nenhum. Ao receber a resposta de que o ID é único, o cliente envia uma mensagem `REQ_CONNSEN` para os dois servidores, dando início à conexão com eles. Essa conexão falha se os servidores já estão tratando o número máximo de clientes simultâneos. A execução do cliente é feita por meio do comando

`./client <IP> <porta_servidor_status> <porta_servidor_localizacao>`.

As portas devem ser passadas nesta ordem, para que os servidores sejam devidamente identificados.

Ao se conectar aos servidores de status e localização, eles geram um status e uma localização aleatórios, respectivamente. O cliente não recebe essas informações como parte do protocolo, mas os comandos descritos logo adiante permitem recuperar esses valores. Uma pequena diferença em relação ao enunciado é que, ao estabelecer e fechar a conexão com um cliente, o servidor de status imprime a localização do cliente em questão. Essa informação não é relevante para o servidor de status e precisaria ser repassada pelo servidor de localização. Logo, o servidor de status imprime o status do cliente que está se conectando ou desconectando.

Com a conexão estabelecida, o cliente possui os seguintes comandos:

1. `kill`: envia `REQ_DISCSEN` para ambos os servidores. Se os servidores o reconhecem, eles imprimem que o cliente se desconectou e o cliente imprime que a desconexão foi bem-sucedida. Caso contrário, o cliente imprime *Sensor not found*. De qualquer forma, o cliente encerra sua execução.
2. `check failure`: verifica se o cliente atual está em estado de alerta (`status` 1), enviando uma mensagem `REQ_SENSSTATUS` ao servidor de status. Se estiver em estado de alerta, o servidor de status requisita a localização deste cliente usando `REQ_CHECKALERT` e retorna este valor ao cliente, por meio de `REs_SENSSTATUS`. O cliente então imprime o estado de alerta, junto com a região onde ele se encontra. Se o cliente atual não estiver registrado nos servidores, ele imprime *Sensor not found* e encerra sua execução.
3. `locate <ID>`: requisita a localização do cliente `ID`. Essa requisição é feita enviando uma mensagem `REQ_SENSLOC` ao servidor de localização, que retorna a localização do cliente por meio de `RES_SENSLOC`. Se o ID não estiver na base de dados dos servidores, o cliente atual imprime *Sensor not found* e continua sua execução normalmente.
4. `diagnose <LocId>`: pede ao servidor de localização uma lista de todos os clientes presentes numa determinada localização `LocId`, por meio de `REQ_LOCLIST`. Se o servidor não reconhece a localização enviada, o cliente imprime *Location not found* e continua sua execução normal. Se não há clientes naquela localização, o cliente imprime *No sensors at location* `LocId` e continua sua execução normal. Se há clientes naquela localização, o cliente imprime *Sensors at location* `LocId`:, seguido dos IDs.

Compilação

O repositório, disponível aqui, contém o `Makefile` na pasta root. Como especificado no enunciado, o comando `make` deve compilar os programas `server` e `client`, sem parâmetros adicionais. Logo, `make` irá compilar os arquivos presentes na pasta `src`. Se necessário, o comando `make clean` remove os dois programas.