LES DE-BUG:
-affichages des choses ou non.

TEST_FOR_ALL_TLV: envoyer régulièrement des tlv de tout type

TEST_FAST_MODE: faire des choses rapidement:
-ajouter le voisin apres un neighbour

MACRO: sur tlv et d'autres choses

---

information courantes:
my_synchronized_number: le 1er seqno retourner au dessus du mien
my_now_synchronized: ce que je devrais avoir comme seqno

---

hashage:
calc_node_hash(): prend des valeurs et fait le [hash d'une "donnee"]

fold_left_tmp_state_hash(): fonctions pour chaque [hash d'une "donnee"], l'ajoute dans [void * sha256context]

get_network_hash(): utilise [fold_left_tmp_state_hash()] pour faire le network hash

---

les tlvs

sizeof_node_state(): donne la reel taille du type 8

---

put_uint16_in_char(): ajoute le uint16_t dans un unsigned char (ded taille au moins 2, dans le bon sens (htons))

fill_struct_addr(): remplir une struct addr, cest tout

get_tlv_type(): retourne le type du tlv ...
get_tlv_len(): sa taille ... (sans compter la tete du tlv qui contient le type et la taille)

tlv_minimum_len_in_expected(): retourne la taille minimal pour un tlv d'un certain type

is_bad_tlv_len_in(): utilise [tlv_minimum_len_in_expected()] et retourne si c'est la bonne taille (assez comme attendu)
(attention: ne retourne pas l'exacte)

compare_node_id(): retourne -1 0 ou 1 (les valeurs sont comme une comparaison classique)

is_same_aux(): est ce que c'est la meme chose: 0 ou 1 (auxiliaire)

is_different_network_hash(): retourne 0 (si pareil) ou 1 (different), pour deux network_hash

compare_seqno(): retourne -1 0 ou 1 (les valeurs sont comme une comparaison classique) [seqno]

is_same_node_id(): retourne 1 si pareil, 0 sinon [node_id]

is_same_node_hash(): retourne 1 si pareil, 0 sinon [node_hash]

predict_next_loss(): essaie de predire le futur changement (a cause d'un pair buguer)

---

print_message(): afficher le message dans le [paquet]

---

convert_4_bit_to_char(): prend un [char] comme si c'etait une valeur de 4 bits (entre 0 et 15 comme valeur)

print_hexa_aux(): auxiliaire: affiche toute la chose en hexadecimal

print_hexa_ip(): hexadecimal de l'ip

print_node_id(): ... node id

print_node_hash(): ...

print_network_hash(): ...

---

fill_network_hash(): remplir un tlv network hash dans la struct [network_hash] (avec le network hash en parametre)

fill_network_state_request(): pareil, pour request

fill_node_state_request(): toujours pareil, pour un [node id] comme parametre

---

fill_pad1(): remplir un pad1 dans [body]

fill_padN(): pareil, pour padN

fill_TLV_in_body(): auxiliaire qui ajoute une struct [un_tlv_au_choix] (avec sa taille en parametre) dans [body]

fill_network_hash_in_body(): dans [body], mettre [network hash]

fill_network_state_request_in_body(): ...

fill_node_hash_in_body(): ...

fill_node_state_in_body(): ...

fill_node_state_request_in_body(): ...

fill_neighbour_in_body(): ...

fill_neighbour_request_in_body(): ...

REMARQUE: PAS DE WARNING ICI (car pas utile?)

---

get_body(): retourne l'emplacement du body (en gros: +4) depuis [head] (le buffer du paquet)

fill_head(): remplir le head (avec taille de ce qui suit (les tlv) en parametre)

fill_fast_head_with_warning(): sa fait [fill_head()] et remplir avec un message en parametre dans [head]

fill_fast_head_with_network_state_request(): pareil, mais avec un [network state request]

fill_fast_head_with_node_state_request(): pareil, ...

fill_fast_head_with_neighbour_request(): ...

---

has_head(): est ce que le buffer contient assez pour avoir une [head] (taille >= 4)

do_not_ignore(): verifie que [magic] == 95 et [version] == 1

get_head_len(): retourne la taille dans le champs [len] du [head] (la taille de tout les autres tlv)

do_not_ignore_with_size(): [has_head()] et [do_not_ignore()] et la taille dans len est correct avec la taille du buffer

---

print_warning(): affiche le message dans un tlv warning
(la limite de l'affichage est WARNING_MSG_SIZE - 1 == 800 - 1) (-1 pour la sentinelle, (protectioncat print un char *))

```
typedef struct memory: la memoire
  unsigned char * node_id;         //tableau des node_id
  uint16_t * seqno;                //tableau des seqno
  char * data;                     //tableau des data
  unsigned char * node_hash;       //tableau des hash
  unsigned char * datalen;         //tableau de la taille de chaque data
  ssize_t * sens_tri;              //tableau qui fait office de permutation,
creer un sens de tri
  ssize_t open;                    //le suivant qui est libre
  int size;                        //la taille total max (courante)
```

init_memory(): creer la memoire

memory_put_node_id_at(): mettre [node id] a l'emplacement i

memory_put_data_and_len_at(): mettre [data] et [datalen] a l'emplacement i

memory_put_node_hash_at(): mettre [node_hash] a l'emplacement i

node_id_at(): retourne l'emplacement du [node id] en i

data_at(): ... [data] ...

node_hash_at(): ... [hash] ...

fold_left_memory(): fonction pour faire un fold right() (haha mauvais nom)

fold_sens_tri_memory(): pareil, mais dans le sens de tri (utile pour faire le [network hash])

struct struct_tmp_remplir_node_state: c'est des infos pour un fold:
```
  int curr_i;                //emplacement courant (dans la boucle)
  int next_i;                //emplacement de celui dont je dois commencer
  ssize_t rest_space_in_octet;     //l'espace restant en octet
  ssize_t put_in_body_in_octet;    //combien on ete ajouter (en octets)
  unsigned char * curr_body;  //l'emplacement courant dans body
```

tmp_fold_remplir_avec_node_state(): c'est la fonction pour un fold, mettre des node state dans [curr_body] du struct precedent

search_node_id_position_in_memory(): c'est la fonction qui recherche de facon dichotomique l'emplacement d'un [node id]
!!! IL EST NON UTILISER !!!

add_new_value_memory(): ajoute une nouvelle valeur dans la memoire (il est supposer: pas de verif)

add_value_memory(): si le [node id] est deja connu: modifie les donnees, sinon [add_new_value_memory()]
!!! IL VERIFIE AUSSI POUR LE NODE COURANT (NOUS) (ET ANTI IMPOSTEUR) !!!

struct struct_fold_node_state: c'est des infos pour un fold:
```
  unsigned char * node_id;          //un node id
  uint16_t * found_seqno;           //le seqno trouver (mettre la valeur dans ce pointeur)
  char * found_data;                //la data trouver
  unsigned char * found_datalen;    //... datalen ...
  unsigned char * found_hash;       //... hash ...
```

fold_node_state_in_memory_tmp(): c'est la fonction pour un fold, essaye de recup les infos du [node_id] dans la struct precedente

get_node_state_with_id_from_memory(): utilise [fold_left_memory()] avec [fold_node_state_in_memory_tmp()]

is_node_id_in_memory(): retourne [DEDANS] ou [PAS_DEDANS] pour un [node id]

node_id_status_in_memory(): retourne [SAME], [NEED] (si on en a besoin), [BETTER] (notre donnee est meilleur), pour un [node id]

fill_node_state_in_body_from_id_in_memory(): remplir un [node state] dans [body] avec un [node id] depuis les infos dans [memory]

fill_fast_node_state_from_id_in_memory(): c'est la meme chose, mais sur [head] (magic, version et la taille pour un seul [node state])

print_memory(): affiche la memoire ...

---

struct voisins
```
  #define NB_VOISIN 15
  int occupe[NB_VOISIN];                 //si c'est occuper ou non (comme sa pas besoin d'effacer/ ...)
  unsigned char ip[IP_SIZE * NB_VOISIN]; //tableau des ip
  uint16_t port[NB_VOISIN];              //WARNING: FORMAT MACHINE (LITTLE ENDIAN)
  int transitoire[NB_VOISIN];            //si c'est transitoire ou non
```

```
  ssize_t last_time[NB_VOISIN];                //la derniere fois que il nous a
envoyer
```

---

get_voisin_char_ip_at(): retourne l'ip d'un voisin en position i (depuis un unsigned char *)

get_voisin_ip_at(): appelle [get_voisin_char_ip_at()] (depuis une struct voisin *)

void init_voisins(struct voisins * v): initialise la struct de voisin (ne le creer pas, sa change les valeurs dans le pointeur)

print_voisins(): affiche la liste des voisins

int ip_port_equals(ip1, port1, ip2, port2): retourne 1 si des ip et port sont egaux, 0 sinon

voisin_add_aux(..., int transit): ajouter un voisin, auxiliare: pour si transitoire ou non ([transit])

voisin_add(): ajoute un voisin, transitoire

voisin_add_permament(): ajoute un voisin non transitoire

voisin_clean(): supprime les voisins qui nous on rien envoyer depuis: #define MAX_TIME 70
(sa change juste [voisin-> occupe[i]] a 0)

voisin_nombre(): retourne le nombre de voisin

voisin_for_all(): fonction de fold, qui applique a tout les voisins

struct struct_fold_random_voisins: struct pour fold
```
  int num;              //un nombre (entre 0 et le nombre de voisins)
  unsigned char * res_ip;     //un ip
  uint16_t * res_port;        //un port
```

get_random_voisins(): retourne un voisin aleatoire

fold_tmp_is_already_in_voisins(): fonction pour un fold

is_already_in_voisins(voisins, ip, port): retourne 1 si [ip] et [port] deja dans [voisins], 0 sinon

struct_fold_show_to_others(): struct pour un fold

fold_tmp_voisin_show_to_others(): fonction tmp pour fold

show_to_others(..., buf, buflen): envoyer un datagramme pour tout les voisins

show_to_others_my_network_hash(): envoyer ... un [network hash]

send_for_all_neighbour_request(): ..... un [neighbour request]

steal_others_network_hash(): .... un [network state request]

send_for_all_node_id_information(): ... un [node state] (depuis un [node id])

magic_send_fast_node_state_request()://c'est pour fonction 2 en bas
magic_send_all_node_hash()://c'est pour fonction 1 en bas
send_for_all_every_tlv_type()//c'est pour lui: envoyer tout les type de tlv (faire des tests)

---

is_all_good_size_message_setter(): tmp pour faire les messages

is_all_good_size(): retourne [ALL_GOOD_SIZE_OK] ou [ALL_GOOD_SIZE_ERR], verifie la coherence des tailles, type de tlv ...

magic_send_fast_warning(): envoyer rapidement un [warning]

magic_send_fast_random_neighbour(): ... [neighbour]

magic_send_fast_network_state_request(): ... [network state request]

magic_send_all_node_hash(): ... [node hash]: tout les nodes hash dans [memory], avec agregation

magic_send_fast_node_state_request(): ... [node state request]

magic_fill_node_state(): il pourrai aussi s'appeller fill_node_state_in_body(): rempli dans [body] un [node state] depuis un [node id] dans [memory]

magic_send_fast_node_state(): envoyer rapidement un [node state] depuis un [node id]

magic_send(): tres inutile ... (pourquoi il n'est pas devant les autres?: c pas grave)

magic_treatment(): il fait tout les traitements des paquets entrant ...
1) verifier avec [is_all_good_size()] (si pas correcte: envoyer [warning] et sortir)
2) ajouter/ mettre a jour, dans la table deds voisins
3) traiter les tlv

PARTICULARITE de magic_treatment():

---

create_new_node_id(): creer un nouveau node id (et ajoute dans le fichier: #define MY_DATA_FILE ".my_node_id.data")

int information_charged = 0;//est ce que au debut: c'est charger (le node id) ou pas (alors charger ou creer)

charge_my_information(): charger le [node id]

add_my_value_in_memory(): ajouter nos valeurs (my_node_id, my_seqno, my_data, my_datalen, hash) dans [memory]

add_my_info_to_memory(..., char * my_data2, ssize_t my_data_len2): ... avec my_data2 et my_data_len2

change_my_information_test(): un test qui change les informations et les ajoutes dans la memoire

print_help_aux(): tmp pour afficher les aides dans le terminal

print_help(): afficher l'aide dans le terminal

super_handler(): s'occupe des entrer dans le terminal

---

main(): il creer bien tout les trucs avec la boucle a evenement