”libgop.a”
Guide de l’utilisateur
Manuel de Référence

Luc Weber

Observatoire de Genève
18 juin 1999

Table des matières

1 Quick Référence **5**

2 Introduction **7**
2.1 Utilisation . . . 7

3 Mode client–serveur, le guide **9**
3.1 Initialisation . . . 9
3.1.1 Initialisation d'un serveur . . . 9
3.1.2 Initialisation d'un client . . . 12
3.1.3 Le niveau de debugging . . . 13
3.2 Envoi d'un message . . . 14
3.2.1 Nom symbolique du destinataire . . . 14
3.2.2 Synchronisation de l'entête . . . 14
3.2.3 Synchronisation des paquets de données . . . 14
3.2.4 Datation du paquet . . . 15
3.2.5 Classe du message . . . 15
3.2.6 Status du système . . . 15
3.3 Réception d'un message . . . 18
3.4 Déconnection . . . 19
3.5 Timeout . . . 20
3.6 Traitement des erreurs . . . 20
3.7 Comportement en cas de déconnection . . . 22

4 Mode multi–client, le guide **23**
4.1 Connection multi–clients multi–ports . . . 23
4.2 Connection multi–clients mono–port . . . 26
4.3 timeout sur select . . . 27

5 Mode transmetteur, le guide **28**

6 Fonctions supplémentaires, le guide **30**
6.1 Envoi de texte . . . 30
6.2 Envoi d'une Fin_De_Message . . . 31
6.3 Réception d'une Fin_De_Message . . . 31
6.4 Communications . . . 32
6.5 Redirection des messages de la librarie . . . 34

7 Gestion des interruptions 36

8 Exemple complet en mode Client–Transmetteur–Serveur 39
8.1 Mode Client–Transmetteur–Serveur: Le Client . . . 40
8.2 Mode Client–Transmetteur–Serveur: Le Transmetteur . . . 44
8.3 Mode Client–Transmetteur–Serveur: Le Serveur . . . 48
8.4 Mode Client–Transmetteur–Serveur: Le fonctions communes . . . 49

9 Manuel de référence, Fonctions principales 52
9.1 gop_accept_connection() . . . 52
9.2 gop_close_connection() . . . 54
9.3 gop_close_init_connection() . . . 55
9.4 gop_close_active_connection() . . . 56
9.5 gop_connection() . . . 57
9.6 gop_forward() . . . 59
9.7 gop_get_error_str() . . . 60
9.8 gop_get_XXXX() . . . 61
9.9 gop_handle_eom() . . . 62
9.10 gop_init_connection() . . . 63
9.11 gop_init_server_socket() . . . 64
9.12 gop_init_client_socket() . . . 64
9.13 gop_init_server_socket_unix() . . . 64
9.14 gop_init_client_socket_unix() . . . 64
9.15 gop_printf() . . . 66
9.16 gop_read() . . . 67
9.17 gop_read_end_of_message() . . . 68
9.18 gop_read_matrix() . . . 69
9.19 gop_registration_for_printf() . . . 70
9.20 gop_select_active_channel() . . . 71
9.21 gop_select_destination() . . . 72
9.22 gop_set_destination() . . . 74
9.23 gop_set_XXXX() . . . 75
9.24 gop_write() . . . 76
9.25 gop_write_acknowledgement() . . . 77
9.26 gop_write_command() . . . 78
9.27 gop_write_end_of_message() . . . 79
9.28 gop_write_matrix() . . . 79

10 Manuel de référence, Fonctions internes 81
10.1 gop_acknow_read() . . . 81
10.2 gop_acknow_write() . . . 81
10.3 gop_alloc_connect_structure() . . . 82
10.4 gop_d_packet_read() . . . 82
10.5 gop_d_packet_write() . . . 83
10.6 gop_data_section_forward() . . . 84
10.7 gop_data_section_read() . . . 84
10.8 gop_data_section_write() . . . 85

10.9 gop_fill_bench_xdr() . . . 86
10.10 gop_header_fill() . . . 86
10.11 gop_first_byte_read() . . . 86
10.12 gop_first_byte_write() . . . 87
10.13 gop_h_read() . . . 88
10.14 gop_header_forward() . . . 88
10.15 gop_header_read() . . . 89
10.16 gop_header_read_without_acknow() . . . 89
10.17 gop_header_write() . . . 90
10.18 gop_io_read() . . . 90
10.19 gop_io_write() . . . 90
10.20 gop_set_struct_standart() . . . 91
10.21 gop_sig_init_handler() . . . 91
10.22 gop_sig_handler() . . . 92
10.23 gop_socket_accept_connection() . . . 92
10.24 gop_socket_connection() . . . 92
10.25 gop_socket_init_connection() . . . 92
10.26 gop_socket_close_connection() . . . 92
10.27 gop_socket_unix_accept_connection() . . . 93
10.28 gop_socket_unix_connection() . . . 93
10.29 gop_socket_unix_init_connection() . . . 93
10.30 gop_socket_unix_close_connection() . . . 93
10.31 gop_socket_read() . . . 94
10.32 gop_socket_write() . . . 94
10.33 gop_test_bench_xdr() . . . 95
10.34 gop_tpu_accept_connection() . . . 96
10.35 gop_tpu_connection() . . . 96
10.36 gop_tpu_init_connection() . . . 96
10.37 gop_tpu_close_connection() . . . 96
10.38 gop_tpu_read() . . . 97
10.39 gop_tpu_write() . . . 97
10.40 gop_update_header() . . . 97

11 Conseils d'utilisations **99**
11.1 Socket Internet entre Suns . . . 99
11.2 Socket Unix entre Suns . . . 99

Chapitre 1

Quick Référence

Initialisation Socket Internet (entre machine)

– **Client**

```
gop_init_client_socket(connect, from, host, port, maxpacket, mode, timeout)
```

– **Server**

```
gop_init_server_socket(connect, from, port, maxpacket, mode, timeout)
```

Initialisation Socket Unix (locale)

– **Client**

```
gop_init_client_socket_unix(connect, from, name, maxpacket, mode, timeout)
```

– **Server**

```
gop_init_server_socket_unix(connect, from, name, maxpacket, mode, timeout)
```

Connection

– **Client**

```
gop_connection(connect)
```

– **Server**

```
gop_init_connection(connect)
gop_accept_connection(connect)
```

Écriture

```
gop_write(connect, data, msize, psize, datatype)
gop_write_acknowledgement(connect, state, texte)
gop_write_command(connect, data)
gop_write_end_of_message(connect, data)
gop_write_matrix(connect, data, msize, psize, datatype, npix_x, dx, dy)
```

Lecture

```
gop_read(connect, buf, sizeof_buf)
gop_read_end_of_message(connect, buf, sizeof_buf)
gop_handle_eom(connect, fct)
gop_read_matrix(connect, cmd, sizeof_cmd, npix_x, dx, dy)
```

transmission

```
gop_forward(from_connect, to_connect)
```

Utilitaires

```
gop_set_destination(connect)
gop_set_XXXX(connect)
gop_get_XXXX(connect)
gop_select_active_channel(list_active, list_ready)
gop_select_destination(from_connect, list, to_connect, buf_size)
```

Déconnection

```
gop_close_connection(connect)
gop_close_init_connection(connect)
gop_close_active_connection(connect)
```

Chapitre 2

Introduction

Le protocole GOP (Geneva Observatory Protocol) permet la communication inter–process pour le passage de messages dans un schéma client–serveur, multi–machines et multi–protocoles.

Un message est composé d'une Entête_De_Message suivit d'une Section_De_Données . L'Entête_De_Message contient tout les paramètre du message sous forme ASCII: taille du message, taille des paquets, modes de synchronisation, classe du message, etc... . La Section_De_Données contient comme son nom l'indique les données du message: chaînes de caractères, vecteurs numériques binaires de tout types.

Un message peut être interrompu en cours de transfert par l'envoi d'un autre message nommé Fin_De_Message qui peut ou non comporter une Section_De_Données .

La synchronisation, qui est facultative, est réalisée par l'envoi de paquets nommés Acknowledge.

Les Entête_De_Message , Section_De_Données , Acknowledge et Fin_De_Message sont composé de paquets. Chaque paquet est précédé d'un caractère qui indique la nature du paquet ('H' pour entête, 'D' pour data, 'A' pour acknowledge et 'E' pour entête de Fin_De_Message). Ce caractère permet de tester la bonne séquence des messages ainsi que l'interruption de transfert avec une Fin_De_Message .

Les messages contenant des données numériques binaires sont automatiquement codés sous le format XDR (eXternal Data Representation) si les interlocuteurs d'un canal de communication ne comprennent pas le même binaire. Remarque: ce codage fait grossir la taille de chaque paquet de la Section_De_Données de 4 bytes. Cet augmentation de taille n'est pas traduite dans le paramètre de l'entête indiquant la taille des paquets (`PSIZE`), mais le message est déclaré de type XDR.

2.1 Utilisation

Les programmes utilisant GOP inclusent dans leur code le fichier `gop.h` qui est actuellement dans le directory `/usr/local/include`.

De plus, GOP supporte actuellement 2 protocoles, TCP/IP sur sockets (Internet ou Unix) et SP pour la communication avec les transputers.

Les programmes n'utilisant pas les transputers se compilent avec la commande:

```
cc prog.c -o prog -I/share/include [options] -lgop -ltpudummy
```

Les programmes utilisant les transputers se compilent avec la commande:

```
cc prog.c -o prog -I/share/include [options] -lgop -ltpu
```

Chapitre 3

Mode client–serveur, le guide

3.1 Initialisation

L'entité de base utilisée par GOP est le canal de communication. Toutes les caractéristiques de ce canal sont stockées dans une structure de type `gop_connect`. Chaque canal a une structure associée. Une telle structure est déclarée de la manière suivante:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
}
```

L'initialisation d'un canal de communication est vue différemment si l'on est un serveur ou un client. Dans notre cas, le serveur est le process qui est l'initiateur de la communication. Le client est le process qui effectue l'opération de connection sur le serveur. Un client peut se connecter sur un serveur uniquement si celui–ci est en attente d'une connection.

3.1.1 Initialisation d'un serveur

L'initialisation d'un canal de communication du côté serveur se réalise à l'aide de deux fonctions:

`gop_init_connection()` Cette fonction initialise un port de communication sur lequel se créera le canal de communication.

`gop_accept_connection()` Cette fonction crée un canal de communication sur un port initialisé. De plus, une fois le canal créé et qu'un client s'est connecté, les deux interlocuteur de ce canal s'échange un message qui permet de fixer certaines caractéristique de ce canal (nom symboliques, taille maximum des paquets, nécessité de XDR).

Pour initialiser un canal de communication, il faut impérativement préciser le type de protocole de transport utilisé. Selon le type de protocole choisi, les paramètres d'initialisations sont différents. On a:

`GOP_SOCKET`	pour les communications TCP/IP sur socket Internet. On donne le numéro de port.
`GOP_SOCKET_UNIX`	pour les communications TCP/IP sur socket Unix. On donne le nom de la socket.
`GOP_TPU`	pour les communications sur transputer de type SP.

Comme les fonctions d'initialisations donnent des informations de debugging, on précise également le niveau de debug à ce moment. Le niveau de debugging sera modifié lors de la réception du premier message, car les messages transportent cette information.
On a donc pour une connection de type socket:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_set_type(&connect, GOP_SOCKET);
   gop_set_mode(&connect, debug_level);
   gop_set_port(&connect, port);

   if (gop_init_connection(&connect) != GOP_OK){...}
   ...
}
```

La création d'un port de communication ne permet pas de communiquer, pour le faire, il faut initialiser le canal de communication. C'est la deuxième phase d'initialisation d'un serveur. De nouveaux paramètres doivent imperativement être précisés avant l'appel à `gop_accept_connection()`. Ce sont:

Le nom symbolique du serveur. Il permettra au client d'associer un nom à la structure de communication.

La taille maximum des paquets. Il permet de définir la taille appropriée à la connection.

L'initialisation complète du serveur devient alors:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_set_type(&connect, GOP_SOCKET);
   gop_set_mode(&connect, debug_level);
   gop_set_port(&connect, port);

   if (gop_init_connection(&connect) != GOP_OK){...}

   gop_set_from(&connect, my_name);
   gop_set_maxpacket(&connect, maxsize);

   if (gop_accept_connection(&connect) != GOP_OK){...}
   ...
}
```

La fonction `gop_accept_connection()` est bloquante, elle se libère uniquement lorsqu'un client se connecte.

L'initialisation de la structure de communication peut être simplifiée par l'appel aux routines d'initialisation spécifiques au type de communication. Pour un serveur basé sur une communication de type socket, on a deux fonctions: `gop_init_server_socket()` et `gop_init_server_socket_unix()`.
L'initialisation ci-dessus devient:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_init_server_socket(&connect, from, port, maxpacket, mode, timeout)

   if (gop_init_connection(&connect) != GOP_OK){...}

   if (gop_accept_connection(&connect) != GOP_OK){...}
   ...
}
```

3.1.2 Initialisation d'un client

L'initialisation d'un client se fait en une seule phase. La connection se fait sur un serveur en attente puis par un échange de message qui permet de fixer certaines caractéristique du canal. Les paramètres que demande la fonction `gop_connection()` dépendent à nouveau du type de protocole de transport choisi. Ce sont pour:

GOP_SOCKET	les mêmes que pour le côté serveur avec en plus le nom de la machine oè se trouve le serveur (avec `gop_set_name()`). Naturelement le numéro de port doit être identique de côté serveur et du côté client.
GOP_SOCKET_UNIX	les mêmes que pour le côté serveur.
GOP_TPU	à définir ....

L'allure d'une connection d'un client de type socket est:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_set_type(&connect, GOP_SOCKET);
   gop_set_name(&connect, host);
   gop_set_port(&connect, port);
   gop_set_maxpacket(&connect, maxsize);
   gop_set_from(&connect, my_name);
   gop_set_mode(&connect, debug_level);

   if (gop_connection(&connect) != GOP_OK){...}
   ...
}
```

Si le serveur n'est pas en attente de connection, la connection est refusée.

L'initialisation de la structure de communication peut être simplifiée par l'appel aux routines d'initialisation spécifiques au type de communication. Pour un client basé sur une communication de type socket, on a deux fonctions: `gop_init_client_socket()` et `gop_init_client_socket_unix()`. L'initialisation ci-dessus devient:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_init_client_socket(&connect, from, host, port, maxpacket,
                          mode, timeout)

   if (gop_connection(&connect) != GOP_OK){...}
   ...
}
```

3.1.3 Le niveau de debugging

Le niveau de debugging tel qu'il est utilisé plus haut avec `gop_set_mode()` défini le niveau de verbosité des fonctions du protocole. Les niveaux se definissent en deux classes.
La première, où plus le niveau de debugging est élevé, plus les informations concernent un partie plus interne du protocole. On a dans ce cas:

`GOP_NOTHING`	pas de message.
`GOP_CONNECTION`	pour les opérations de connection et de select.
`GOP_MESSAGE`	pour la réception ou l'envoi de messages.
`GOP_MESSAGE_HEADER`	pour l'affichage de l'entête.
`GOP_PACKET`	pour la réception ou l'envoi de paquets.
`GOP_PACKET_INFO`	pour des informations concernant les paquets.
`GOP_IO`	pour les opérations d'entrée–sortie bas niveau.
`GOP_IO_CONTENTS`	pour le contenu des buffers transitant en entrée–sortie.

La deuxième classe donne deux niveaux de debugging uniques:

`GOP_HEADER`	pour l'affichage des entêtes.
`GOP_HEADER_CONTENTS`	pour l'affichage des entêtes et du contenus des messages de type `GOP_CHAR`.

3.2 Envoi d'un message

L'envoi de messages permet de faire transiter entre un client et un serveur des tableaux de données de type reconnus par GOP de n'importe quelle taille. Avant l'envoi d'un message les paramètres suivants sont à mettre à jour:

3.2.1 Nom symbolique du destinataire

Définis avec `gop_set_to()` le nom du destinataire.

3.2.2 Synchronisation de l'entête

Définis, avec `gop_set_hsync()`, si l'entête doit être aquitée. On a deux possibilités:

`GOP_SYNCHRO`	avec quittance.
`GOP_ASYNCHRO`	sans quittance.

La quittance est le seul moyen qui permet de récupérer un éventuel status d'erreur dans le cas où: l'entête n'est pas acceptée (mauvaise version), si le serveur ne peut pas recevoir de tableau de taille trop élevée, etc...

3.2.3 Synchronisation des paquets de données

Définis, avec `gop_set_dsync()`, si chaque paquets formant la Section_De_Données doit être acquitté. On a deux possibilités:

`GOP_SYNCHRO`	avec quittance.
`GOP_ASYNCHRO`	sans quittance.

La synchronisation à pour but de régulariser le transfert des donnés. Normallement, ce type de synchronisation n'est pas nécessaire avec les sockets car le protocole TCP/IP gère très efficacement ce type de synchronisation.

3.2.4 Datation du paquet

Définis, avec `gop_set_stamp()`, si la date de l'entête doit est mise. On a deux possibilités:

`GOP_TRUE`	pour mettre la date.
`GOP_FALSE`	pour ne pas mettre la date.

3.2.5 Classe du message

Définis avec `gop_set_class()` la classe du message. Cette information permet à l'interlocuteur de trier les messages qui lui sont addressé. Les classe sont:

`GOP_CLASS_COMD`	pour les commandes.
`GOP_CLASS_DATA`	pour les données.
`GOP_CLASS_STAT`	pour le status.
`GOP_CLASS_INFO`	pour les informations.
`GOP_CLASS_DBUG`	pour le debugging.
`GOP_CLASS_ACKN`	pour les quittances.
`GOP_CLASS_ALRM`	pour les alarmes.

3.2.6 Status du système

Définis avec `gop_set_stat()` l'état du système. Cette information permet à l'interlocuteur de connaître l'état du système qui à envoyé le message. Les classe sont:

GOP_STAT_OPOK	système ok.
GOP_STAT_WARN	avertissement.
GOP_STAT_RCOV	erreur récupérable.
GOP_STAT_FTAL	erreur fatale.
GOP_STAT_BUSY	système est occupé.
GOP_STAT_TIME	timeout.

Ainsi pour envoyer par exemple un tableau de données sur un client nommé "inter", en mode synchrone sur l'entête et asynchrone sur les données depuis un système OK avec la date dans l'entête, on initialisera l'envoi du message avec:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_set_class(&connect, GOP_CLASS_DATA);
   gop_set_stat(&connect, GOP_STAT_OPOK);
   gop_set_stamp(&connect, GOP_TRUE);
   gop_set_hsync(&connect, GOP_SYNCHRO);
   gop_set_dsync(&connect, GOP_ASYNCHRO);
   gop_set_to(&connect, "inter");
   ...
}
```

L'envoi des données se fait avec la fonction `gop_write()` où l'on précise la taille du tableau, la taille des paquets et le type de donnée.
Les types de données reconnus sont:

`GOP_CHAR`	8 bits.
`GOP_USHORT`	unsigned int 16 bits.
`GOP_SHORT`	int 16 bits.
`GOP_UINT`	unsigned int 32 bits.
`GOP_INT`	int 32 bits.
`GOP_ULONG`	unsigned int 64 bits.
`GOP_LONG`	int 64 bits.
`GOP_FLOAT`	real 32 bits.
`GOP_DOUBLE`	real 64 bits.

Si le canal de communication a besoin d'une convertion de type XDR, le protocole utilise ce paramètre pour la convertion. Il n'y a aucune convertion pour les données de type `GOP_CHAR`.

Le code permettant l'envoi d'un tableau nommé `buf` de 1000 entiers 16 bits non–signés par groupe de 100 élements est le suivant:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   gop_set_class(&connect, GOP_CLASS_DATA);
   gop_set_stat(&connect, GOP_STAT_OPOK);
   gop_set_stamp(&connect, GOP_TRUE);
   gop_set_hsync(&connect, GOP_SYNCHRO);
   gop_set_dsync(&connect, GOP_ASYNCHRO);
   gop_set_to(&connect, "inter");

   if (gop_write(&connect, buf, 2000, 200, GOP_USHORT) != GOP_OK){...}
   ...
}
```

Il faut toutefois remarquer que la taille des paquets sera réduite si on donne à `gop_write()` une taille de paquet supérieur à la taille maximum autorisée sur le canal (voir plus haut `gop_set_maxpacket()`).

3.3 Réception d'un message

Un process qui reçoit un message sait à priori quel type de donnée il attend. La fonction `gop_read()` demande donc uniquement l'adresse d'un tableau et la taille de celui–ci.
Le code qui permet par exemple la lecture du tableau précédent ressemble à:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ushort buf[10000];
   int    ilen;
   ...
   if ((ilen = gop_read(&connect, buf, sizeof(buf)) < 0){...};
   ...
}
```

La fonction `gop_read()` retourne le nombre de bytes effectivement lu (dans notre exemple, `ilen` vaudra 2000).
GOP permet à l'expéditeur de raccourcir la longueur des messages en cours de transmission. C'est à dire qu'un process qui par exemple s'attend à lire 2000 bytes peut en recevoir effectivement 1000. Dans ce genre de situation, la valeur retournée par `gop_read()` est négative et sa valeur absolue indique le nombre de bytes effectivement lu. De plus `gop_errno` est posé à `GOP_END_OF_MESSAGE` voir plus bas sous: "réception d'une Fin_De_Message ".
La récupération de la classe, du status, de la date, etc... se font avec les fonctions `gop_get_XXXX`. Par exemple:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   char class[5], stat[5], date[17];
   ...
   if ((ilen = gop_read(&connect, buf, sizeof(buf)) < 0){...};
   class = gop_get_class(&connect);
   stat = gop_get_stat(&connect);
   date = gop_get_date(&connect);
   ...
}
```

3.4 Déconnection

Un canal de communication se déconnecte completement avec la fonction `gop_close_connection()`. On peut déconnecter uniquement la socket active (connect.cd) avec `gop_close_connection()` ou uniquement la socket d'initialisation (connect.cd_init) avec `gop_close_active_connection()`. Par exemple:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   if (gop_close_connection(&connect) != GOP_OK){...}
   /* ou */
   if (gop_close_active_connection(&connect) != GOP_OK){...}
   if (gop_close_init_connection(&connect) != GOP_OK){...}
   ...
}
```

3.5 Timeout

Toutes les fonctions qui à un moment donné effectuent la lecture d'un paquet, peuvent le faire avec un timeout. Ce cas se présente dans toute les fonctions de lecture, mais aussi dans les fonctions d'écriture où l'on synchronise les messages avec la lecture d'un paquet d'Acknowledge.

L'utilisation du timeout se décide en posant la valeur `timeout` de la structure du canal de communication, sur lequel a lieu l'acces, à une valeur différente de zéro. Cette valeur donne le temps d'attente maximum pour une lecture en secondes (valeurs entières).
Une fonction qui se termine sur un timeout retourne `GOP_KO` et pose `gop_errno` à `GOP_TIMEOUT`.

Exemple:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   connect->timeout = 30;
   if (gop_read(&connect, buf, sizeof(buf)) != GOP_OK){
      if(gop_errno == GOP_TIMEOUT){
         ...
      } else {
         ...
      }
  }
}
```

3.6 Traitement des erreurs

La majorité des fonctions de la librarie GOP retournent un status. Si aucun problème ne c'est produit une fonction retourne `GOP_OK`. Si par contre, une erreur est survenue la fonction retourne `GOP_KO`. Le numéro de l'erreur est à recupérer dans la variable globale `gop_errno`, la fonction `gop_get_error_str()` permet de récupérer le texte sommaire concernant l'explication de l'erreur.

Par exemple:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   ...
   if (gop_init_connection(&connect) != GOP_OK){
      fprintf(stderr, "Erreur GOP : gop_init_connection: %s\n",
                      gop_get_error_str());
      exit();
   }
   ...
}
```

La liste des code d'erreur est la suivante:

GOP_ERRNO	erreur système dont le code est dans errno.
GOP_DISCONNECT	déconnection du client.
GOP_INVALID_VERSION	version de l'entête invalide.
GOP_TIMEOUT	time out.
GOP_TOBIG	message trop grand.
GOP_BAD_PROTOCOL	protocole de transport inconnu.
GOP_NOT_IMPLEMENTED	protocole de transport pas encore implémenté.
GOP_BROKEN_PIPE	broken pipe lors d'un IO.
GOP_BAD_SEQUENCE	réception d'un bloc illégal.
GOP_RECEIVER_UNKNOWN	destinataire inconnu.
GOP_END_OF_MESSAGE	fin de message prématuré.
GOP_ALLOC	problème d'allocation mémoire.
GOP_BAD_CHANNEL	mauvais canal d'initialisation.
GOP_XDR_FAILED	problème avec convertion XDR.
GOP_REMOTE_PROBLEM	problème côté destinataire.

3.7 Comportement en cas de déconnection

Durant une communication, lorsqu'un interlocuteur est tué, son partenaire termine sur une erreur lors de l'accès suivant.
Par exemple, lorsqu'un client est tué, le serveur (en attente sur un read()) se termine dans tout les cas par l'erreur: `GOP_DISCONNECT`.
Par contre lorsqu'un serveur est tué, le client se termine de façon différente selon qu'il effectue un `write()` ou un `read()` et selon le protocole utilisé. C'est à dire:

- Avec les sockets unix, dans tout les cas le client se termine sur une erreur `GOP_BROKEN_PIPE`.
- Avec les sockets internet, lors d'un `write()`, le client se termine sur une erreur `GOP_BROKEN_PIPE`.
- Avec les sockets internet, lors d'un `read()` sous `BSD`, le client se termine sur une erreur `GOP_DISCONNECT`.
- Avec les sockets internet, lors d'un `read()` sous `System V`, le client se termine sur une erreur `GOP_ERRNO` avec `errno=ECONNRESET`.

Par exemple le problème sur le `write()` à lieu lors d'un `gop_write_command()` sans synchronisation (au moment de l'envoi de la Section_De_Données , le premier `write()` de la Entête_De_Message se passe toujours sans erreur), tandis que le problème sur le `read()` à lieu lors d'un `gop_write_command()` avec synchronisation (au moment de la lecture de l'acknowledge)
Si l'on souhaite gérer ce type de problème, il faut initialiser un handler de signaux pour SIGPIPE. Voir section "signaux" et la fonction `gop_sig_init_handler()`.

Chapitre 4

Mode multi–client, le guide

Un serveur peut posséder plusieurs canaux de communications vers différents clients. Le serveur peut travailler avec un port par client ou avec un port pour tous les clients (avec le type `GOP_SOCKET`, pour le cas d'un log–book par exemple). Quelque soit le type de communication, une difficulté apparaît si l'on autorise la connection des clients dans un ordre aléatoire. La fonction `gop_select_active_channel()` permet de gérer ce cas. Elle demande en entrée une liste de canaux sur lequels une connection peut survenir (canaux qui ont passé la première phase de leur initialisation avec `gop_init_connection()`).

4.1 Connection multi–clients multi–ports

L'exemple suivant montre la fabrication d'une liste (de type `gop_list`) de trois canaux.

```
#include <gop.h>

main()
{
   struct gop_connect connect_client_a;
   struct gop_connect connect_client_b;
   struct gop_connect connect_client_c;
   struct gop_list    input_list, output_list;
   ...
   if (gop_init_connection(&connect_client_a) != GOP_OK){...}
   if (gop_init_connection(&connect_client_b) != GOP_OK){...}
   if (gop_init_connection(&connect_client_c) != GOP_OK){...}

   input_list.nb=3;
   input_list.gop[0] = &connect_client_a;
   input_list.gop[1] = &connect_client_b;
   input_list.gop[2] = &connect_client_c;
   ...
}
```

Cette liste est passée à `gop_select_active_channel()` qui se met en attente de connection et qui retourne une liste de cannaux sur lequels un client c'est manifesté:

```
#include <gop.h>

main()
{
   struct gop_list    input_list, output_list;
   ...
   input_list.nb=3;
   input_list.gop[0] = &connect_client_a;
   ...
   gop_select_active_channel(&input_list, &output_list)
   ...
}
```

Cette liste doit être balayée et chaque canal de `output_list` doit être connecté. La liste doit être reformée (en reduisant le nombre de client) et `gop_select_active_channel()` doit être renvoyée jusqu'à que tout les client soit connectés. La connection s'écrit dans ce cas:

```
#include <gop.h>

main()
{
   struct gop_list    input_list, output_list;
   ...
   gop_select_active_channel(input_list, output_list)
    for (i = 0; i < output_list.nb; i++) {
      gop_accept_connection(output_list.gop[i]);
      ...
   }
}
```

Il faut toutefois prendre garde qu'un client peut commencer à envoyer des messages avant que tout les clients soient connectés. Ce cas est pris en charge par GOP qui sait reconnaître une structure de communication (`gop_connect`) en attente de connection d'une structure de communication en attente de message. En effet, la structure contient le numéro du "chanel descriptor" (`cd`). Celui-ci vaut `-1` tant que le canal n'est pas connecté.

Ainsi, si l'on veut gérer ce cas globalement, on garde une liste stable et l'action à entreprendre après un `gop_select_active_channel()` dépend de la valeur de `cd`.

Le traitement global devient par exemple:

```
#include <gop.h>

main()
{
   struct gop_list    input_list, output_list;
   ...
   input_list.nb=3;
   input_list.gop[0] = &connect_client_a;
   ...
   while(TRUE){
      gop_select_active_channel(&input_list, &output_list)
      for (i = 0; i < output_list.nb; i++) {
         if(gop_get_cd(output_list.gop[i])==-1) {
            gop_accept_connection(output_list.gop[i]);
         } else {
            gop_read(output_list.gop[i], buf, sizeof(buf));
            ...
         }
      }
   }
}
```

4.2 Connection multi–clients mono–port

C'est le cas typique d'un utilitaire de type log–book. Un nombre indéfini de clients peuvent se connecter sur un port unique et y envoyer des messages. Comme on l'a vu dans l'exemple précédent, GOP reconnaît un structure de communication en attente de connection si le "channel descriptor" `cd` vaut `-1`. Une fois connecté, `cd` à une valeur supérieur ou égale à `0`.
Dans le cas suivant, il est donc nécessaire de conserver une structure pour l'attente de connection et de créer une structure pour chaque nouveau client.
L'exemple suivant montre ce type d'application:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   struct gop_list    input_list, output_list;
   ...
   if (gop_init_connection(&connect) != GOP_OK){...}
   input_list.nb     = 1;
   input_list.gop[0] = &connect;

   while (TRUE) {
      if(gop_select_active_channel(&input_list, &output_list)!=GOP_OK)
         {...}
      for (i = 0; i < output_list.nb; i++) {
         if (gop_get_cd(output_list.gop[i]) == -1) {
            input_list.gop[input_list.nb] = (struct gop_connect *)
                                  malloc(sizeof(struct gop_connect));
            if (input_list.gop[input_list.nb] == NULL) {...}
            memcpy(input_list.gop[input_list.nb], output_list.gop[i],
                                 sizeof(struct gop_connect));

            gop_accept_connection(input_list.gop[input_list.nb]);

            input_list.nb = input_list.nb + 1;
         } else {
            if(gop_read(output_list.gop[i], buf, sizeof(buf)) < 0)
               {...}
            printf("recu >%s<\n", buf);
            ....
         }
      }
   }
}
```

4.3 timeout sur select

La fonction `gop_select_active_channel()` peut travailler avec un timeout. Dans ce cas c'est la valeur de `input_list->timeout` qui est pris en compte. Si cette valeur est différente de zéro, la fonction travaille avec un timeout (exprimé en secondes entières), sinon, non.

Chapitre 5

Mode transmetteur, le guide

Le mode Client–Transmetteur–Serveur permet à un client d'adresser un serveur par l'intermédiaire d'un transmetteur. Ce schéma d'adressage est utilisé par exemple lorsque le client n'a pas la possibilité d'avoir un canal de communication direct avec le serveur (pas de protocole unique entre le serveur et le client). Le transmetteur a de cette façon les mêmes caractéristiques qu'un serveur et peut ainsi soit recevoir des messages qui lui sont destinés, soit transmettre les messages destiné au serveur.

L'alternative à la fonction `gop_read()` est `gop_select_destination()`. Cette fonction lit uniquement l'Entête_De_Message d'un message et determine le canal de communication du destinataire d'après une liste de canaux actif. Si le message est destiné au process qui lancé `gop_select_destination()`, le canal de destination est retourné comme `NULL`. Sinon cette fonction retourne un des canaux de la liste.

Pour un message addressé au transmetteur, la Section_De_Données du message reste donc à lire avec `gop_data_section_read()` (l'Entête_De_Message ayant déjà été lue par `gop_select_destination()`). La fonction `gop_data_section_read()` retourne les mêmes valeurs que `gop_read()`.

Par contre si le message est destiné au serveur, il faut transmettre l'Entête_De_Message et la Section_De_Données vers le destinataire final avec `gop_forward()`.

L'exemple suivant montre le code d'un programme travaillant en mode transmetteur:

```
#include <gop.h>

main()
{
   struct gop_connect   connect_client;
   struct gop_connect   connect_server;
   struct gop_connect   *to_connect;
   ...
   list.nb = 1;
   list.gop[0] = &connect_server;
   gop_select_destination(&connect_client, &list, &to_connect, sizeof(cmd));

   if (to_connect == NULL) {
      /* on est le destinataire */
      if (gop_data_section_read(&connect_client, cmd, sizeof(cmd)) < 0) {...}
      ...
   } else {
      /* on est pas le destinataire */
     gop_forward(&connect_client, to_connect);
   }
   ...
}
```

Chapitre 6

Fonctions supplémentaires, le guide

Quelques fonctions supplémentaires existent pour simplifier la programation.

6.1 Envoi de texte

La fonction `gop_write_command()` facilite l'envoi d'une chaîne de caractère. L'exemple suivant montre l'appel standard équivalent et l'appel simplifié:

```
main()
{
   struct gop_connect   connect;
   char                 cmd[80];
   ...
   if(gop_write(&connect, cmd, strlen(cmd)+1, maxpacket, GOP_CHAR) != GOP_OK)
     {...}
}
```

devient

```
main()
{
   struct gop_connect   connect;
   char                 cmd[80];
   ...
   if(gop_write_command(&connect, cmd) != GOP_OK)
     {...}
}
```

6.2 Envoi d'une Fin_De_Message

Un client peut décider d'interrompre l'envoi d'un message en cours. Par exemple, on décide d'interrompre un message envoyé paquet par paquet, dont le nombre total de paquets est inconnu (interaction avec l'utilisateur par exemple).
Pour faire cela, le client ne peut pas utiliser `gop_write` mais doit utiliser les appels de base qui le compose.
Dans l'exemple suivant le client envoi un message fourni par une fonction `xxxx()` qui retourne une valeur `GOP_FALSE` indiquant si le message est le dernier. Le serveur s'attend au maximum a 100 messages et affichera le message envoyé avec le Fin_De_Message .

```
#include <gop.h>

main()
{
   struct gop_connect   connect;
   char                 cmd[100]
   ...
   gop_set_msize = 100*sizeof(cmd);
   gop_set_psize = sizeof(cmd);
   gop_set_datatype = GOP_CHAR;
   if(gop_header_fill(&connect)!=GOP_OK){...}
   if(gop_header_write(&connect)!=GOP_OK){...}

   while (xxxx(cmd)){
      if(gop_d_packet_write(&connect, cmd, sizeof(cmd))!=GOP_OK){...}
   }
   if(gop_write_end_of_message(&connect, "fin d'acquisation")!=GOP_OK){...}
   ...
}
```

6.3 Réception d'une Fin_De_Message

L'exemple précédent est maintenant vu du côté serveur. Le message est lu avec la fonction `gop_read()` qui se termine sur une erreur `GOP_END_OF_MESSAGE`. Le contenu de Fin_De_Message est lu avec la fonction `gop_read_end_of_message()` qui est capable de lire un message dont le 1er byte de l'Entête_De_Message à déjè été lu (cas de terminaison de message standard).
Attention, la fonction `gop_read()` retourne un nombre négatif pour indiquer que la communication a été interrompue par une Fin_De_Message . La valeur absolue de ce nombre indique le nombre de bytes effectivement lus. Si cette valeur vaut `-1`, cela indique non pas qu'un byte à été lu mais qu'une erreur c'est produite. Le code suivant indique la manière de traiter nomrmalement ce cas.

```
#include <gop.h>

main()
{
   struct gop_connect   connect;
   char                 buf[10000], text[100];
   ...
   if((len = gop_read(&connect, buf, sizeof(buf))< 0){
      if (gop_errno == GOP_END_OF_MESSAGE) {
         if (gop_read_end_of_message(&connect_client, text, sizeof(text)) < 0)
              {...}
         fprintf(stderr, "MESSAGE: >%s<\n", text);
         if (len == -1)len = 0;
         len = abs(len);
      } else {
         ... /* erreur communication */
      }
   ...
}
```

6.4 Communications

Une communication regroupe un ensemble de messages liés les uns aux autres.
Par exemple un client qui veut envoyer des données, envoye habituellement une commande au préhalable. Dans ce cas, le client envoye ces 2 messages sous la forme d'une communication unique, comprenant la commande puis les données.
Le moyen d'indiquer à un process (travaillant en mode transmetteur) qu'un message est lié à un autre et donc que ce process doit rester en écoute du même canal jusqu'à la fin de la communication est donné par le paramètre `CONT` de l'Entête_De_Message . Ce paramètre indique par `GOP_TRUE` que le message courant est lié au message suivant.

L'exemple suivant montre un client envoyant une communication comprenant une commande et des données.

```
#include <gop.h>

main()
{
   struct gop_connect   connect;
   ...
   gop_set_cont(&connect, GOP_TRUE);
   if(gop_write_command(&connect, cmd) != GOP_OK)
     {...}
   gop_set_cont(&connect, GOP_FALSE);
   if (gop_write(&connect, buf, msize, psize, data_type) != GOP_OK)
     {...}
   ...
}
```

Le serveur (en mode transmetteur) a cette allure:

```
#include <gop.h>

main()
{
   struct gop_connect   connect_client;
   struct gop_connect   connect_server;
   struct gop_connect   *to_connect;
   ...
   list.nb = 1;
   list.gop[0] = &connect_server;
   gop_select_destination(&connect_client, &list, &to_connect, sizeof(cmd));

   if (to_connect == NULL) {
      /* on est le destinataire */
      if (gop_data_section_read(&connect_client, cmd, sizeof(cmd)) < 0) {...}
      ...
   } else {
      /* on est pas le destinataire */
     gop_forward(&connect_client, to_connect);
     while(gop_get_cont(&connect)){
        if(gop_h_read(&connect)!=GOP_OK){...}
        if(gop_header_read(&connect)!=GOP_OK){...}
        if(gop_forward(&connect_client, to_connect)!=GOP_OK){...}
     }
   }
   ...
}
```

Dans l'exemple précédent on considère l'utilisation de `CONT` pour un serveur en mode transmetteur. En effet, si le serveur est le destinataire final, le decodage de la commande qui lui est adressée lui indique que des données suivent la commande. Et donc ce type de serveur ne se soucie pas de la valeur de `CONT`.

6.5 Redirection des messages de la librarie

Les messages de la librarie sortent par défaut sur `stdout`. Il est possible de rediriger ces messages vers toutes autres sortie en enregistrant une fonction utilisateur. Cette fonction reçoit en argument la chaîne formatée et elle peut la renvoyer par exemple dans un fichier, sur le logbook sur `stdout` ou encore simultanement sur plusieurs sorties possibles.
Exemple:

```
print_on_logbook(str)
        char            *str;
{
        char             out[256];

        sprintf(out, "my_name: %s", str);

        if (log_book_status != -1) {
                log_book_status = lbk_write_log_book(out);
        } else {
                print_on_stdout(out);
        }
}


{...
        gop_registration_for_printf(print_on_logbook);
...}
```

Chapitre 7

Gestion des interruptions

GOP supporte l'interruption d'une communication chez le client. Le principe est simple:

- Si le client est interrompu (CTRL–C) lors d'un write (`gop_write()`), il envoie un bloc Fin_De_Message à la place d'un bloc standard et la communication s'interrompt.
- Si le client est interrompu lors d'un read (`gop_read()`), il transmet un signal SIGURG au serveur (Out Of Band data sur socket internet ou kill() sur socket unix). A ce moment, le serveur envoie un bloc Fin_De_Message à la place d'un bloc standard et la communication s'interrompt.

Il est donc vital que le process qui recoit l'interruption sache s'il joue le rôle de client ou de serveur. Par défaut, le serveur est le processus qui initialise la connection avec `gop_init_connection()` et `gop_accept_connection()` et le client est le processus qui initialise la connection avec `gop_connection()`. Si les rôles sont interchangés ou si le processus est un transmetteur, une assignation du rôle doit être faite après la connection. Cette assignation se fait avec la fonction: `gop_set_side()` à laquelle on fourni comme argument: le pointeur sur la structure de connection et le rôle selon les valeurs suivantes:

`GOP_SERVER_SIDE`	joue le rôle de serveur.
`GOP_CLIENT_SIDE`	joue le rôle de client.
`GOP_TRANSMIT_SIDE`	joue le rôle de transmetteur.

GOP installe ses propres handlers de signaux durant chaque communication (SIGINT, SIGURG, SIGPIPE et SIGALRM). Si le processus possède déjà ces handlers, GOP les exécutera durant le traitement du signal et les réinstalle en fin de communication.

Remarque importante: un CTRL–C durant une interruption n'interrompt donc plus le processus. Si ce comportement est désiré, il faut installer un handler spécifique. Deplus chaque interruption en cours de communication (côté client) fait terminer la communication par un Fin_De_Message . Il est donc **indispensable** de détecter l'erreur `GOP_END_OF_MESSAGE` après un `gop_read()` et dans le cas échant effectuer un `gop_read_end_of_message` si ce n'est pas fait, gop est desynchroniser et il faut alors jouer sur les timeout.

Exemple:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   char   buf[100000];
   /* ... */
   if (gop_read(&connect, buf, sizeof(buf)) < 0) {
      if (gop_errno == GOP_END_OF_MESSAGE) {
         if (gop_read_end_of_message(&connect, buf, sizeof(buf)) < 0) {
            manage_gop_error("gop_read_second");
         }
         printf("Recu fin de message: >%s<\n", buf);
      } else {
         manage_gop_error("gop_read");
      }
   }
   /* ... */
}
```

ou, avec l'utilisation `gop_handle_eom()`:

```
#include <gop.h>

main()
{
   struct gop_connect connect;
   char   buf[100000];
   /* ... */
   if (gop_read(&connect, buf, sizeof(buf)) < 0) {
         gop_handle_eom(&connect, NULL);
         manage_gop_error("gop_read");
   }
   /* ... */
}
```

De son côté, si un client envoie une commande à un serveur et reste en attente de données en retour, une interruption peut survenir pendant l'attente du client sur les données alors que le serveur n'as pas encore commencé le `gop_write()` permettant de rapatrier les données. Dans ce cas le serveur recoit un signal SIGURG qui n'est pas géré par GOP car le serveur n'est pas entrain de faire une communication. Le serveur doit donc posséder un handler de SIGURG et mémoriser si un interrupt a eu lieu. Dans le cas échant, le serveur doit explicitement envoyer une Fin_De_Message pour terminer la communication avec le client.
Exemple:

```
#include <gop.h>

static int server_interrupt;

void handler_sigurg(sig)
   int            sig;
{
   server_interrupt = 1;
}

main()
{
   struct gop_connect connect;
   char           buf[100000];
   /* ... */
   if (gop_read(&connect, buf, sizeof(buf)) < 0) {   /* ... */ }
   server_interrupt = 0;
   /* ... processing ... */
   if (server_interrupt) {
      if(gop_write_end_of_message(&connect,
                                  "Interrupted transmission")!= GOP_OK)
         { /* ... */ }
   } else {
      if (gop_write(&connect, buf, sizeof(buf), 4096, GOP_CHAR) != GOP_OK)
         { /* ... */ }
   }
   /* ... */
}
```

Chapitre 8

Exemple complet en mode Client–Transmetteur–Serveur

L'exemple suivant montre un aspect typique d'une transmission client serveur. Le client envoie une commande (sous forme ASCII) puis envoie ou recupère des données.
Dans cet exemple, le client et le serveur travaillent par défaut sur des sockets unix (type:`GOP_SOCKET_UNIX`) nommées `"socket.server"` ou peuvent travailler avec des sockets internet (type:`GOP_SOCKET`) avec l'option `-h` <`hostname`> donnée au sur le client. Dans ce cas le numéro de port est fixé à 1280.
On peut interposer un transmetteur entre le client et le serveur avec l'option `-t` donnée au client . Dans ce cas le transmetteur travaille par défaut sur les sockets nommées `"socket.transmit"` ou sur le port 1281. Le transmetteur se connecte sur le serveur de la même manière que le client en mode client–serveur.
Le client peut accéder soit le serveur, soit le transmetteur. Le serveur et le transmetteur se terminent si le client est tué.

Voir les fichiers sous `~weber/exemples/gop: client.c transmitter.c server.c`

8.1 Mode Client–Transmetteur–Serveur: Le Client

```
#include <stdio.h>
#include <gop.h>
#include <signal.h>
#include <demo.h>

static int      client_interrupt;

static void
handler_ctrlc(sig)
   int             sig;
{
   signal(SIGINT, handler_ctrlc);
   printf("handler_ctrlc: recu SIGINT \n");
   client_interrupt = 1;
}

main(argc, argv)
   int             argc;
   char          **argv;
{
   struct gop_connect connect;
   char            answer[128];
   char            host_name[80];
   char            socket_name[10];
   int             port = 1280, maxpacket = 4096, mode = GOP_HEADER_CONTENTS;
   int             socket_unix = GOP_TRUE, transmitter = GOP_FALSE;
   char            from[10];
   int             flag, status;

   extern char    *optarg;
   char            c;

   signal(SIGINT, handler_ctrlc);


   strcpy(socket_name, "server");
   strcpy(from, *argv);

   while ((c = getopt(argc, argv, "h:tm:")) != -1) {
      switch (c) {
      case 'h':
         strcpy(host_name, optarg);
         socket_unix = GOP_FALSE;
         break;
      case 't':
```

```
        transmitter = GOP_TRUE;
        port = 1281;
        strcpy(socket_name, "transmit");
        break;
     case 'm':
        sscanf(optarg, "%d", &mode);
        break;
     default:
        fprintf(stderr, "Options: [-h <host_name>] [-t] [-m] <mode>\n");
        exit(0);
     }
  }

  if (socket_unix) {
     gop_init_client_socket_unix(&connect, from, socket_name,
                 maxpacket, mode, 0);
  } else {

     gop_init_client_socket(&connect, from, host_name,
                 port, maxpacket, mode, 0);
  }
  gop_set_stamp(&connect, GOP_TRUE);

  if (gop_connection(&connect) != GOP_OK) {
     manage_gop_error("gop_connection");
     exit(0);
  }
  gop_set_to(&connect, "server");

  for (;;) {
     client_interrupt = 0;
     printf("TAPEZ r(ead)  lecture  de 500 [KB] en provenance du serveur\n");
     printf("      w(rite) écriture de 500 [KB] sur le serveur\n");
     printf("      R(EAD)  lecture  de 500 [KB] en provenance du transmetteur\n");
     printf("      W(RITE) écriture de 500 [KB] sur le transmetteur\n");
     printf("      si la 2eme lettre est un 'm' le transfert est en mode matrix\n");
     printf("      0-9     choix du niveau de debug\n");
     printf("      q(uit)  pour quitter:\n");
     gets(answer);

     if (*answer == 'q')
        exit(0);

     if (*answer >= '0' && *answer <= '9')
        gop_set_mode(&connect, atoi(answer));

     if (client_interrupt == 0 && (*answer == 'r' || *answer == 'w' ||
```

```
            *answer == 'R' || *answer == 'W')) {

         if (transmitter && (*answer == 'R' || *answer == 'W'))
            gop_set_to(&connect, "transmit");
         else
            gop_set_to(&connect, "server");

         printf("ENVOI COMMAND SUR >%s<\n", gop_get_to(&connect));

         gop_set_class(&connect, GOP_CLASS_COMD);
         gop_set_cont(&connect, GOP_TRUE);
         gop_set_stat(&connect, GOP_STAT_OPOK);

         *answer = *answer | 0x20;
         if (*(answer + 1) != 0)
            *(answer + 1) = *(answer + 1) | 0x20;
         if (gop_write_command(&connect, answer) != GOP_OK)
            manage_gop_error("gop_write_command");

         flag = *(answer + 1) == 'm' || *(answer + 1) == 'M';

         if (*answer == 'r') {
            status = read_data(&connect, flag);
         } else {
            status = write_data_client(&connect, flag);
         }
         if (status < 0)
            manage_gop_error("gop_write_command");
      }
   }
}

int
write_data_client(connect, flag)
   struct gop_connect *connect;
   int             flag;
{
   int             buf[NPIX_X * NPIX_Y];
   int             i;
   int             status;

   /* envoi des data */

   printf("\n\n Préparation data ... \n");
   for (i = 0; i < sizeof(buf) / sizeof(int); i++)
      buf[i] = i;
```

```
   printf(" Envoi de data..... \n\n");
   gop_set_cont(connect, GOP_FALSE);
   gop_set_class(connect, GOP_CLASS_DATA);

   if (flag)
      status = gop_write_matrix(connect, (char *) &buf, XSIZE * YSIZE * sizeof(int),
               XSIZE * sizeof(int), GOP_INT, NPIX_X, DX, DY);
   else
      status = gop_write(connect, (char *) &buf, sizeof(buf), 4096, GOP_INT);


   if (status != GOP_OK) {
      if (gop_errno == GOP_INTERRUPTED_TRANSMISSION) {
         printf("message incomplet\n");
      } else {
         manage_gop_error("gop_read");
      }
   }
   return (status);
}
```

8.2 Mode Client–Transmetteur–Serveur: Le Transmetteur

```
#include <stdio.h>
#include <signal.h>
#include <gop.h>
#include <demo.h>

static int      server_interrupt;

static void
handler_sigurg(sig)
   int             sig;
{
   signal(SIGURG, handler_sigurg);
   fprintf(stderr, "handler_sigurg: recu   SIGURG (OOB) \n");
   server_interrupt = 1;
}

main(argc, argv)
   int             argc;
   char          **argv;
{
   struct gop_connect connect_client_inet;
   struct gop_connect connect_client_unix;
   struct gop_connect connect_server;
   struct gop_connect *to_connect;
   struct gop_list active_list;
   struct gop_list input_list, output_list;

   char            cmd[128];
   int             i, flag, status;
   char            host_name[80];
   char            socket_name_server[] = "server";
   char            socket_name_client[] = "transmit";
   int             port_server = 1280, port_client = 1281;
   int             maxpacket = 4096, mode = GOP_CONNECTION;
   int             socket_unix = GOP_TRUE;
   char            from[] = "transmit";

   extern char    *optarg;
   char            c;


   signal(SIGURG, handler_sigurg);


   while ((c = getopt(argc, argv, "h:")) != -1) {
```

```
      switch (c) {
      case 'h':
         strcpy(host_name, optarg);
         socket_unix = GOP_FALSE;
         break;
      default:
         fprintf(stderr, "Options: [-h <host_name>]\n");
         exit(0);
      }
   }

   if (socket_unix) {
      gop_init_client_socket_unix(&connect_server, from, socket_name_server,
                   maxpacket, mode, 0);
   } else {

      gop_init_client_socket(&connect_server, from, host_name,
                   port_server, maxpacket, mode, 0);
   }
   gop_set_stamp(&connect_server, GOP_TRUE);

   if (gop_connection(&connect_server) != GOP_OK) {
      manage_gop_error("gop_connection");
      exit(0);
   }
   gop_init_server_socket(&connect_client_inet, from, port_client,
                             maxpacket, mode, 0);
   gop_init_server_socket_unix(&connect_client_unix, from,
                             socket_name_client, maxpacket, mode, 0);
   gop_set_stamp(&connect_client_inet, GOP_TRUE);
   gop_set_stamp(&connect_client_unix, GOP_TRUE);

   if (gop_init_connection(&connect_client_inet) != GOP_OK) {
      manage_gop_error("init sur socket internet");
      exit(0);
   }
   if (gop_init_connection(&connect_client_unix) != GOP_OK) {
      manage_gop_error("init sur socket unix");
      exit(0);
   }
   input_list.timeout = 0;
   input_list.nb = 3;
   input_list.gop[0] = &connect_client_inet;
   input_list.gop[1] = &connect_client_unix;
   input_list.gop[2] = &connect_server;

   active_list.nb = 1;
```

```
active_list.gop[0] = &connect_server;


while (1) {
   if (gop_select_active_channel(&input_list, &output_list) != GOP_OK) {
      manage_gop_error("gop_select_active_channel");
   }
   for (i = 0; i < output_list.nb; i++) {

      if (gop_get_cd(output_list.gop[i]) == -1) {
         if (gop_accept_connection(output_list.gop[i]) != GOP_OK) {
            manage_gop_error("gop_accept_connection");
            exit(0);
         }
         active_list.gop[active_list.nb] = output_list.gop[i];
         active_list.nb = active_list.nb + 1;
      } else {

         if (gop_select_destination(output_list.gop[i], &active_list,
                  &to_connect) != GOP_OK) {
            manage_gop_error("gop_select_destination");
         }
         if (to_connect != NULL) {
            printf("COMMUNICATION EN TRANSIT\n");

            gop_set_side(output_list.gop[i], GOP_TRANSMIT_SIDE);

            if (gop_forward(output_list.gop[i], to_connect) != GOP_OK) {
               manage_gop_error("gop_read");
            }
         } else {

            printf("COMMUNICATION LOCALE\n");
            gop_set_side(output_list.gop[i], GOP_SERVER_SIDE);
            if (gop_read_data(output_list.gop[i], cmd, sizeof(cmd)) < 0) {
               manage_gop_error("gop_read");
            }
            printf("\n\n Recu: >%s<\n\n", cmd);
            flag = *(cmd + 1) == 'm';
            if (*cmd == 'r') {
               status = write_data_server(output_list.gop[i], flag);
            } else {
               status = read_data(output_list.gop[i], flag);
            }
            if (status < GOP_OK)
               manage_gop_error("gop_write_command");
```

```
                }
            }
        }
    }
}
```

8.3 Mode Client–Transmetteur–Serveur: Le Serveur

```
#include <stdio.h>
#include <signal.h>
#include <gop.h>
#include <demo.h>

static int      server_interrupt;

static void
handler_sigurg(sig)
   int             sig;
{
   signal(SIGURG, handler_sigurg);
   fprintf(stderr, "handler_sigurg: recu   SIGURG (OOB) \n");
   server_interrupt = 1;
}

main()
{
   struct gop_connect connect_unix;
   struct gop_connect connect_inet;
   struct gop_list input_list, output_list;

   char            cmd[128];
   int             i, flag, status;
   int             port = 1280, maxpacket = 1540, mode = GOP_CONNECTION;
   int             timeout = 0;
   char            from[] = "server", socket_name[] = "server";

   signal(SIGURG, handler_sigurg);

   gop_init_server_socket(&connect_inet, from, port, maxpacket,
                                mode, timeout);
   gop_init_server_socket_unix(&connect_unix, from, socket_name,
                                 maxpacket, mode, timeout);
   gop_set_stamp(&connect_inet, GOP_TRUE);
   gop_set_stamp(&connect_unix, GOP_TRUE);

   if (gop_init_connection(&connect_inet) != GOP_OK) {
      manage_gop_error("init sur socket internet");
      exit(0);
   }
   if (gop_init_connection(&connect_unix) != GOP_OK) {
      manage_gop_error("init sur socket unix");
      exit(0);
   }
```

```
   input_list.timeout = 0;
   input_list.nb = 2;
   input_list.gop[0] = &connect_inet;
   input_list.gop[1] = &connect_unix;

   while (1) {
      if (gop_select_active_channel(&input_list, &output_list) != GOP_OK) {
         manage_gop_error("gop_select_active_channel");
      }
      for (i = 0; i < output_list.nb; i++) {
         if (gop_get_cd(output_list.gop[i]) == -1) {
            if (gop_accept_connection(output_list.gop[i]) != GOP_OK) {
               manage_gop_error("gop_accept_connection");
               exit(0);
            }
         } else {
            if (gop_read(output_list.gop[i], cmd, sizeof(cmd)) < 0) {
               manage_gop_error("gop_read");
            }
            printf("\n\n Recu: >%s<\n\n", cmd);
            flag = *(cmd + 1) == 'm';
            if (*cmd == 'r') {
               status = write_data_server(output_list.gop[i], flag);
            } else {
               status = read_data(output_list.gop[i], flag);
            }
            if (status < GOP_OK)
               manage_gop_error("gop_write_command");

         }
      }
   }
}
```

8.4 Mode Client–Transmetteur–Serveur: Le fonctions communes

```
static void
manage_gop_error(txt)
   char          *txt;
{
   fprintf(stderr, "manage_gop_error: Erreur GOP : %s: %s\n", txt, gop_get_error_str());
   if (gop_errno == GOP_DISCONNECT)
      exit(0);
}
```

```
int
read_data(connect, flag)
   struct gop_connect *connect;
   int              flag;
{
   int              buf[NPIX_X * NPIX_Y];
   int              status;

   if (flag) {
      if ((status = gop_read_matrix(connect, (char *) &buf, sizeof(buf),
                 NPIX_X, DX, DY)) < 0)
         gop_handle_eom(connect, NULL);
   } else {
      if ((status = gop_read(connect, (char *) &buf, sizeof(buf))) < 0)
         gop_handle_eom(connect, NULL);
   }

   return (status);
}

int
write_data_server(connect, flag)
   struct gop_connect *connect;
   int              flag;
{
   int              buf[NPIX_X * NPIX_Y];
   int              i;
   int              status;

   server_interrupt = 0;

   /* envoi des data */

   printf("\n\n Préparation data  \n");
   for (i = 0; i < sizeof(buf) / sizeof(int); i++)
      buf[i] = i;

   for (i = 0; i < 4; i++) {
      sleep(1);
      printf(".");
      fflush(stdout);
      if (server_interrupt)
         break;
   }
   printf("\n");

   gop_set_destination(connect);
```

```
  if (server_interrupt) {
     /*
      * envoi de EOM si le serveur a ete interrompue hors
      * communication
      */
     printf("!!!! Interrupt Détecté durant la péparation des data: \n");
     printf("!!!!    ->  envoi de fin de message\n");
     gop_write_end_of_message(connect, "Interrupted transmission");
     return (GOP_KO);

  } else {
     /*
      * envoi des data
      */
     printf(" Envoi data  \n\n");

     gop_set_cont(connect, GOP_FALSE);
     gop_set_class(connect, GOP_CLASS_DATA);
     if (flag)
        status = gop_write_matrix(connect, (char *) &buf, XSIZE * YSIZE * sizeof(int),
              XSIZE * sizeof(int), GOP_INT, NPIX_X, DX, DY);
     else
        status = gop_write(connect, (char *) &buf, sizeof(buf), 4096, GOP_INT);

     if (status != GOP_OK)
        return (GOP_KO);

  }
  return (GOP_OK);
}
```

Chapitre 9

Manuel de référence, Fonctions principales

9.1 gop_accept_connection()

Etabli une connection côté serveur sur un port qui a déjà été initialisé avec `gop_init_connection()`.

Synopsis:

```
int
gop_accept_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Une fois `gop_accept_connection()` lancé, un client peut se connecter avec `gop_connection()`.

Lorsque la connection est établie, `gop_accept_connection()` attend un message en provenance de son interlocuteur. Ce message est envoyé automatiquement par `gop_connection()`.

L'Entête_De_Message doit contenir dans le champ `header.psize` la taille maximum que le client est prêt à recevoir et le nom symbolique du client dans le champ `header.from`.

`gop_accept_connection()` determine la taille maximum des paquets pour cette connection selon le minimum de `connect->maxpacket` et `connect->header.psize`. Il met à jour `connect->to` avec `connect->header.from`.

La Section_De_Données contient des constantes de chaque type numérique reconnu par GOP (short, int, long, float et double). Ces constantes permettent de déterminer automatiquement si le canal de communication nécessite ou non l'utilisation de XDR.

`gop_accept_connection()` met à jour `connect->need_xdr`.

Une fois les données interprétées, `gop_accept_connection()` envoie à son tour les mêmes indication à son interlocuteur, c'est à dire la taille maximum des paquets en réception, le nom symbolique du serveur et les constantes.

Variables devant être initialisées avant l'appel à cette fonction:

connect−>type	type de connection.
connect−>cd_init	channel descriptor initial.
connect−>from	nom symbolique de l'expéditeur.
connect−>maxpacket	taille maximum des paquets sur `cd` en bytes.
connect−>mode	niveau de debug.

Variables mises à jour par cette commande:

connect−>cd	channel descriptor actif.
connect−>to	nom symbolique du destinataire.
connect−>need_xdr	indique si le cannal a besoin du codage en XDR
connect−>maxpacket	taille maximum des paquets sur `cd` en bytes.
connect−>class	Classe du message. Posé à `GOP_CLASS_COMD.`
connect−>cont	flag de continuation Posé à `GOP_FALSE.`
connect−>stamp	flag pour écrire la date dans le Entête_De_Message . Posé à `GOP_TRUE.`
connect−>hsync	Flag pour la synchronisation du Entête_De_Message . Posé à `GOP_SYNCHRO.`
connect−>dsync	Flag pour la synchronisation de la Section_De_Données . Posé à `GOP_SYNCHRO.`
connect−>stat	status du système. Posé à `GOP_STAT_OPOK`
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR.`

Exemple:

```
struct gop_connect   connect;
...
if(gop_init_connection(&connect) != GOP_OK){...}
if(gop_accept_connection(&connect) != GOP_OK){...}
```

9.2 gop_close_connection()

Ferme completement un canal de communication (cd +cd_init).

Synopsis:

```
int
gop_close_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect structure associée au canal de communication

9.3 gop_close_init_connection()

Ferme la partie initialisation d'un canal de communication (cd_init).

Synopsis:

```
int
gop_close_init_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

9.4 gop_close_active_connection()

Ferme la partie active d'un canal de communication (cd).

Synopsis:

```
int
gop_close_active_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect structure associée au canal de communication

9.5 gop_connection()

Etabli une connection côté client.

Synopsis:

```
int
gop_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

`gop_connection()` se connecte sur un serveur qui a effectué un `gop_accept_connection()`.

Lorsque la connection est établie, `gop_connection()` envoie un message à son interlocuteur.

Le header du message doit contenir dans le champ `connect->header.psize` la taille maximum que le client est prêt à recevoir ainsi que le nom symbolique du client dans le champ `connect->header.from`.

La partie data contient des constantes de chaque type numérique reconnu par GOP (short, int, long, float et double).

Une fois envoyé, il se met en attente d'un message, en provenance du serveur, contenant les mêmes indications.

`gop_connection()` determine la taille maximum des paquets pour cette connection selon le minimum de `connect->maxpacket` et `connect->header.psize`. Il met à jour `connect->to` avec `connect->header.from`. Les constantes reçus dans la partie data permettent de déterminer automatiquement si le canal de communication nécessite ou non l'utilisation de XDR. `gop_connection()` met à jour `connect->need_xdr`.

Variables devant être initialisées avant l'appel à cette fonction:

connect−>type	type de connection.
connect−>from	nom symbolique de l'expéditeur.
connect−>maxpacket	taille maximum des paquets sur `cd` en bytes.
connect−>mode	niveau de debug.
SOCKET:	
connect−>name	nom lié à la connection.
connect−>port	port de communication.
SOCKET_UNIX:	
connect−>name	nom lié à la connection.
TPU:	

Variables mises à jour par cette commande:

connect−>cd	channel descriptor actif.
connect−>to	nom symbolique du destinataire.
connect−>need_xdr	indique si le cannal a besoin du codage en XDR
connect−>maxpacket	taille maximum des paquets sur `cd` en bytes.
connect−>class	Classe du message. Posé à `GOP_CLASS_COMD`.
connect−>cont	flag de continuation Posé à `GOP_FALSE`.
connect−>stamp	flag pour écrire la date dans le Entête_De_Message . Posé à `GOP_TRUE`.
connect−>hsync	Flag pour la synchronisation du Entête_De_Message . Posé à `GOP_SYNCHRO`.
connect−>dsync	Flag pour la synchronisation de la Section_De_Données . Posé à `GOP_SYNCHRO`.
connect−>stat	status du système. Posé à `GOP_STAT_OPOK`
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR`.

Exemple:

```
struct gop_connect   connect;
if(gop_connection(&connect) != GOP_OK){...}
```

9.6 gop_forward()

Transmet un message complet (Entête_De_Message + Section_De_Données)

Synopsis:

```
int
gop_forward(from_connect, to_connect)
    struct gop_connect *connect
```

Paramètres:

from_connect	structure associée à un canal de communication en réception
to_connect	structure associée à un canal de communication en émission

---

Description:

L'utilisation de cette fonction se fait après que l'Entête_De_Message ait été lu. Ainsi l'Entête_De_Message est transmis puis la Section_De_Données est réceptionée et envoyée paquet par paquet. (voir `gop_header_forward()` et `gop_data_section_forward()`.

Exemple:

```
struct gop_connect   from_connect;
struct gop_connect   *to_connect;
...
gop_select_destination(&connect_client, &list, &to_connect, size);
if (to_connect == NULL) {
   /* on est le destinataire */
   ....
} else {
   /* on est pas le destinataire */
  gop_forward(&connect_client, to_connect);
}
```

9.7 gop_get_error_str()

Retourne le texte du message d'erreur donné par `gop_errno`.

Synopsis:

```
char *
gop_get_error_str()
```

9.8 gop_get_XXXX()

Lecture d'un élément d'une structure de type gop_connect

Synopsis:

```
type_of_XXXX
gop_get_XXXX(connect)
    struct gop_connect *connect
```

Paramètres:

connect	structure associée au canal de communication

Description:

Les élément accessibles par cette serie de fonctions sont:

class	avec: gop_get_class(connect)
cd	avec: gop_get_cd(connect)
cont	avec: gop_get_cont(connect)
date	avec: gop_get_date(connect)
datatype	avec: gop_get_datatype(connect)
dsync	avec: gop_get_dsync(connect)
from	avec: gop_get_from(connect)
name	avec: gop_get_name(connect)
hsync	avec: gop_get_hsync(connect)
maxpacket	avec: gop_get_maxpacket(connect)
mode	avec: gop_get_mode(connect)
msize	avec: gop_get_msize(connect)
port	avec: gop_get_port(connect)
psize	avec: gop_get_psize(connect)
side	avec: gop_get_side(connect)
stamp	avec: gop_get_stamp(connect)
stat	avec: gop_get_stat(connect)
timeout	avec: gop_get_timeout(connect)
to	avec: gop_get_to(connect)
type	avec: gop_get_type(connect)

9.9 gop_handle_eom()

Fonction d'aide au traitement du Fin_De_Message

Synopsis:

```
void
gop_handle_eom(connect, fct)
    struct gop_connect *connect
    void               (*fct) ();
```

Paramètre:

connect	structure associée au canal de communication
status	status courant
fct	fonction de gestion d'erreur

Description:

Cette fonction est principalement utilisée après un `gop_read()` et permet de gérer la Fin_De_Message si le cas se présente (si le `gop_read()` se termine sur un `GOP_END_OF_MESSAGE`). Ce traitement permet de récupérer un texte de 256 caractères au maximum et de le passer à la fonction `fct()`. Si cette fonction est déclarée `NULL`, le message est affiché à l'écran. `gop_errno` vaut `GOP_INTERRUPTED_TRANSMISSION` s'il valait `GOP_END_OF_MESSAGE` ou vaut `GOP_EOM_TOO_BIG` s'il y a eu un `GOP_END_OF_MESSAGE` et que le message est plus grand que 256 caractères ou si les 256 caractères du message n'ont pas pu être alloués (malloc).

Exemple:

```
my_error_handler(txt)
          char           *txt;
{
          fprintf(stderr, "Erreur GOP : %s: %s\n", txt, gop_get_error_str());
          if (gop_errno == GOP_DISCONNECT)
                    exit();
}

main()
{
          /* ... */
          if ((status=gop_read(&connect, buf, sizeof(buf))) < 0) {
                    gop_handle_eom(&connect, my_error_handler);
                    my_error_handler("gop_read");
          }
          /* ... */
}
```

9.10 gop_init_connection()

Initialise un canal de communication côté serveur

Synopsis:

```
int
gop_init_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

`gop_init_connection()` est la première phase d'initialisation du coté serveur, cette fonction doit être suivie par `gop_accept_connection()`.

Variables devant être initialisées avant l'appel à cette fonction:

connect−>type	type de connection.
connect−>mode	niveau de debug.
SOCKET:	
connect−>port	port de communication.
SOCKET_UNIX:	
connect−>name	nom lié à la connection.
TPU:	

Variables mises à jour par cette commande:

SOCKET:	
SOCKET_UNIX:	
connect−>cd_init	channel descriptor initial.
connect−>cd	channel descriptor actif. `connect->cd` est posé à -1.
TPU:	
connect−>cd_init	channel descriptor initial. `connect->cd_init` est posé à -1.
connect−>cd	channel descriptor actif. `connect->cd` est posé à -1.

9.11 gop_init_server_socket()

9.12 gop_init_client_socket()

9.13 gop_init_server_socket_unix()

9.14 gop_init_client_socket_unix()

Fonction d'initialisation spécifique au type de connection.

Synopsis:

```
void
gop_init_server_socket(connect, from, port, maxpacket, mode, timeout)

void
gop_init_client_socket(connect, from, host, port, maxpacket, mode, timeout)

void
gop_init_server_socket_unix(connect, from, name, maxpacket, mode, timeout)

void
gop_init_client_socket_unix(connect, from, name, maxpacket, mode, timeout)

    struct gop_connect *connect;
    char               *from;
    char               *host;
    int                port;
    int                maxpacket;
    int                mode;
    int                timeout;
```

Paramètre:

connect	structure associée au canal de communication
from	nom symbolique du prgramme lançanat cette fonction
host	nom internet de la machine serveur
name	nom du socket (sans le path) pour `GOP_SOCKET`
port	No de port pour `GOP_SOCKET_UNIX`
maxpacket	taille maximum des paquets en réception
mode	niveau de verbosité du protocole
timeout	timeout pour les opération d'entrée–sortie

Description:

Ces fonctions remplacent les suites d'appel aux fonctions `gop_set_xxx()` utilisées pour initialiser les structure de communications.

Exemple:

```
gop_init_client_socket_unix(connect, 'aff', 'aff', 2048, GOP_NOTHING, 0);
```

9.15 gop_printf()

Fonction d'impression utilisée par la librairie. Cette fonction s'utilise comme le `printf()`, mais est traitée par la fonction enregistrée par `gop_registration_for_printf()`

Synopsis:

```
int
gop_printf(format, args, ...)
    char              *format;
```

Paramètres:

format	format (voir printf())
args	arguments de tout types liés à ce format

9.16 gop_read()

Lecture d'un message complet dont on se sait le destinataire

Synopsis:

```
int
gop_read(connect, buf, sizeof_buf)
    struct gop_connect *connect
    char               *buf
    int                 sizeof_buf
```

Paramètres:

connect	structure associée au canal de communication
buf	tableau de destination
sizeof_buf	taille du tableau

Description:

Retourne le nombre de bytes effectivement lu (voir `gop_data_section_read()`).

Exemple:

```
struct gop_connect   connect;
char  cmd[128];

gop_init_connection(&connect);
gop_accept_connection(&connect);

gop_read(&connect, cmd, sizeof(cmd));
.....
```

9.17 gop_read_end_of_message()

Comme `gop_read()`, mais sans la lecture du premier byte d'Entête_De_Message .

Synopsis:

```
int
gop_read_end_of_message(connect, buf, sizeof_buf)
    struct gop_connect *connect
    char               *buf
    int                 sizeof_buf
```

Paramètres:

connect	structure associée au canal de communication
buf	tableau de destination
sizeof_buf	taille du tableau

Description:

Utilisé pour lire une Fin_De_Message alors que le premier byte ('E') à déjà été lu. Par `gop_data_section_read()` par exemple.

Exemple:

```
struct gop_connect   connect;
char  cmd[128];

.....
if(gop_errno == GOP_END_OF_MESSAGE){
   gop_read_end_of_message(&connect, cmd, sizeof(cmd));
   .....
}
```

9.18 gop_read_matrix()

Lecture de données dont la destination est une matrice incluse dans une matrice de taille supérieure ou égale.

Synopsis:

```
int
gop_read_matrix(connect, buf, sizeof_buf, npix_x, dx, dy)
    struct gop_connect *connect;
    char               *buf;
    int                 sizeof_buf;
    int                 npix_x;
    int                 dx;
    int                 dy;
```

Paramètres:

connect	structure associée au canal de communication
buf	tableau de destination
sizeof_buf	taille totale du tableau en bytes
npix_x	taille du tableau selon X en pixel
dx	offset du premier pixel selon X en pixel
dy	offset du premier pixel selon Y en pixel

Description:

Utilisé pour lire des parties d'image. `dx` et `dy` sont donnés selon l'origine de la matrice `buf` avec une origine en <0;0>.

9.19 gop_registration_for_printf()

Permet d'enregistrer un fonction utilisateur pour la redirection des message de la librairie.

Synopsis:

```
int
gop_registration_for_printf(fct)
    void              (*fct) ();
```

Paramètres:

fct	fonction utilisateur

Description:

La fonction reçoit la chaîne formatée et l'imprime sur la sortie de son choix.

9.20 gop_select_active_channel()

Se met en attente d'un message sur un canal actif.

Synopsis:

```
int
gop_select_active_channel(list_active, list_ready)
    struct gop_list *list_active;
    struct gop_list *list_ready;
```

Paramètres:

list_active	liste des cannaux de communications actifs.
list_ready	liste des cannaux ayant un message en attente.

Description:

Cette fonction se met en attente aussi bien pour des réceptions de messages que pour des demandes de connections.

Pour une structure `gop_connect` dont l'adresse est donnée dans `liste_active` l'indication s'il s'agit d'une attente en connection ou d'une attente de message est donée par la variable `connect->cd`. Si elle vaut `-1`, cela signifie que le canal n'est pas actif. Dans ce cas, le select se fait sur `connect->cd_init`. Si `connect->cd` est différent de `-1`, c'est sur cette valeur que ce fera le select.

Une fois le `select` terminé, `gop_select_active_channel()` retourne la liste des cannaux en attente dans `list_ready`. Cette liste peut contenir plus d'un élément car plusieurs clients ont pu faire un accès avant que le serveur ne lance `gop_select_active_channel()` ou alors qu'il était occupé. Dans ce cas la priorité (position dans la liste) est conservée de la liste d'origine `liste_active`.

Si `connect->timeout` du premier canal de communication de `list_active` est différent de zéro, la fonction travaille avec un timeout (exprimé en secondes entières).

9.21 gop_select_destination()

Lecture de Entête_De_Message et determination du destinataire.

Synopsis:

```
int
gop_select_destination(from_connect, list, to_connect, buf_size)
    struct gop_connect  *connect
    struct gop_list     *list;
    struct gop_connect **to_connect;
    int                  buf_size
```

Paramètres:

from_connect	structure associée à un canal de communication en réception
list	liste des connections actives
to_connect	structure associée à un canal de communication en émission
buf_size	taille du buffer pour les communications locales

Description:

`gop_select_destination()` lit l'Entête_De_Message . La compatibilité de version est testée et la structure `from_connect` est mise à jour selon les indications du header.

Si `list` est valide (pas égal à `NULL`) cela signifie que le programme exécutant `gop_select_destination()` peut travailler en mode transmetteur. Dans ce cas `list` contient la liste des connections actives.

Si le programme exécutant `gop_select_destination()` est reconnu comme destinataire, `to_connect` est posé à `NULL`.

Si le programme exécutant `gop_select_destination()` n'est pas reconnu comme destinataire, `to_connect` est mis à jour et pointe sur la structure reconnue comme destinatrice. Dans ce cas l'Entête_De_Message est copié integralement dans `connect->header`.

Exemple:

```
struct gop_connect   connect_a;
struct gop_connect   connect_b;
struct gop_connect   connect_c;
struct gop_connect   *to_connect;
...
list.nb = 3;
list.gop[0] = &connect_a;
list.gop[1] = &connect_b;
list.gop[2] = &connect_c;
...
gop_select_destination(&connect_a, &list, &to_connect, size);

if (to_connect == NULL) {
   /* on est le destinataire */
   ...
} else {
   /* on est pas le destinataire */
  gop_forward(&connect_a, to_connect);
}
...
```

9.22 gop_set_destination()

Inverse les noms de l'expéditeur et du destinataire avant l'envoi d'un message vers l'expéditeur du message courant.

Synopsis:

```
void
gop_set_destination(connect)
    struct gop_connect *connect;
```

Paramètres:

connect	structure associée au canal de communication

9.23 gop_set_XXXX()

Mis à jour d'un élément d'une structure de type gop_connect

Synopsis:

```
void
gop_set_XXXX(connect, XXXX)
    struct gop_connect *connect
    type_of_XXXX        XXXX
```

Paramètres:

connect	structure associée au canal de communication
XXXX	élément d'une structure de type gop_connect

Description:

Les élément accessibles par cette serie de fonctions sont:

class	avec: gop_set_class(connect, class)
cont	avec: gop_set_cont(connect, cont)
datatype	avec: gop_set_datatype(connect, datatype)
dsync	avec: gop_set_dsync(connect, dsync)
from	avec: gop_set_from(connect, from)
name	avec: gop_set_name(connect, name)
his_name	avec: gop_his_my_name(connect, his_name)
hsync	avec: gop_set_hsync(connect, hsync)
maxpacket	avec: gop_set_maxpacket(connect, maxpacket)
mode	avec: gop_set_mode(connect, mode)
msize	avec: gop_set_msize(connect, msize)
my_name	avec: gop_set_my_name(connect, my_name)
port	avec: gop_set_port(connect, port)
psize	avec: gop_set_psize(connect, psize)
side	avec: gop_set_side(connect, side)
stamp	avec: gop_set_stamp(connect, stamp)
stat	avec: gop_set_stat(connect, stat)
timeout	avec: gop_set_timeout(connect, timeout)
to	avec: gop_set_to(connect, to)
type	avec: gop_set_type(connect, type)

9.24 gop_write()

Envoi complet d'un message (Entête_De_Message + Section_De_Données).

Synopsis:

```
int
gop_write(connect, data, msize, psize, datatype)
    struct gop_connect *connect
    char               *data
    int                 msize
    int                 psize
    int                 datatype
```

Paramètres:

connect	structure associée au canal de communication
buf	tableau
msize	taille du message
psize	taille des packet
datatype	type de donnée

Variables mises à jour par cette commande:

connect−>msize	taille message en bytes. Posé à `msize`.
connect−>psize	taille paquet en bytes. Posé à `psize`.
connect−>datatype	type de données dans la Section_De_Données . Posé à `datatype`.

Exemple:

```
struct gop_connect   connect;
char  cmd[128];

gop_init_connection(&connect);
gop_accept_connection(&connect);

strcpy(cmd,"commande");
gop_write(&connect, cmd, sizeof(cmd), connect.maxpacket, GOP_CHAR);
.....
```

9.25 gop_write_acknowledgement()

Envoi complet d'un message (Entête_De_Message + Section_De_Données) de classe ACKN.

Synopsis:

```
int
gop_write_acknowledgement(connect, state, texte)
    struct gop_connect *connect
    char               *state
    char               *texte
```

Paramètres:

connect	structure associée au canal de communication
state	état du système (voir `GOP_STAT_XXXX`
texte	tableau

Description:

Cette fonction accepte des chaîne de caractères de longueur nulle.

Variables mises à jour par cette commande:

connect−>class	Classe du message. Posé à `GOP_CLASS_ACKN`.
connect−>stat	status du système. Posé à `stat`.
connect−>msize	taille message en bytes. Posé à `sizeof(texte)`.
connect−>psize	taille paquet en bytes. Posé à `connect->maxpacket`.
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR`.

Exemple:

```
gop_write_acknowledgement(&connect, GOP_STAT_RCOV, "Erreur synchro");
...
gop_write_acknowledgement(&connect, GOP_STAT_OPOK, "");
```

9.26 gop_write_command()

Envoi complet d'un message (Entête_De_Message + Section_De_Données) de classe COMD.

Synopsis:

```
int
gop_write_command(connect, data)
    struct gop_connect *connect
    char               *data
```

Paramètres:

connect	structure associée au canal de communication
data	tableau

Variables mises à jour par cette commande:

connect−>msize	taille message en bytes. Posé à `strlen(data)+1`.
connect−>psize	taille paquet en bytes. Posé à `connect->maxpacket`.
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR`.

Exemple:

```
struct gop_connect   connect;
char  cmd[128];

gop_init_connection(&connect);
gop_accept_connection(&connect);

strcpy(cmd,"initialisation");
gop_write_command(&connect, cmd);
.....
```

9.27 gop_write_end_of_message()

Envoi complet d'un Fin_De_Message .

Synopsis:

```
int
gop_write_end_of_message(connect, data)
    struct gop_connect *connect
    char               *data
```

Paramètres:

connect	structure associée au canal de communication
data	tableau

Variables mises à jour par cette commande:

connect−>msize	taille message en bytes. Posé à `sizeof(data)`.
connect−>psize	taille paquet en bytes. Posé à `connect->maxpacket`.
connect−>hsync	Flag pour la synchronisation du Entête_De_Message . Posé à `GOP_SYNCHRO`.
connect−>dsync	Flag pour la synchronisation de la Section_De_Données . Posé à `GOP_SYNCHRO`.
connect−>cont	flag de continuation Posé à `GOP_FALSE`.
connect−>stamp	flag pour écrire la date dans le Entête_De_Message . Posé à `GOP_TRUE`.
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR`.

Exemple:

```
struct gop_connect   connect;
char  cmd[128];

gop_init_connection(&connect);
gop_accept_connection(&connect);

strcpy(cmd,"fin de données");
gop_write_end_of_message(&connect, cmd);
.....
```

9.28 gop_write_matrix()

Envoi de données dont la source est une matrice incluse dans une matrice de taille supérieure ou égale.

Synopsis:

```
int
gop_write_matrix(connect, buf, msize, psize, datatype, npix_x, dx, dy)
    struct gop_connect *connect;
    char               *buf;
    int                 sizeof_buf;
    int                 npix_x;
    int                 dx;
    int                 dy;
```

Paramètres:

connect	structure associée au canal de communication
buf	tableau de destination
msize	taille de la matrice à transferer en bytes
psize	nombre de bytes dans une ligne de cette matrice
datatype	type de données
npix_x	taille du tableau selon X en pixel
dx	offset du premier pixel selon X en pixel
dy	offset du premier pixel selon Y en pixel

Description:

Utilisé pour envoyer des parties d'image. `dx` et `dy` sont donnés selon l'origine de la matrice `buf` avec une origine en <0;0>.

Chapitre 10

Manuel de référence, Fonctions internes

10.1 gop_acknow_read()

Lit un packet de type acknowledge

Synopsis:

```
int
gop_acknow_read(connect, remote_status)
    struct gop_connect *connect
    int                *remote_status
```

Paramètre:

connect	structure associée au canal de communication
remote_status	status de l'interlocuteur

Description:

Retourne dans `remote_status` le status d'erreur de l'interlocuteur.

Si ce status est différent de zéro, `gop_acknow_read()` retourne `GOP_KO`.

10.2 gop_acknow_write()

Écrit un packet de type acknowledge

Synopsis:

```
int
gop_acknow_write(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Code la valeur de `gop_errno` dans le packet acknowledge.

10.3 gop_alloc_connect_structure()

allocation d'un structure de communication

Synopsis:

```
struct gop_connect *
gop_alloc_connect_structure()
```

Exemple:

```
program fgop
integer  connect_struct, status

call fortran_alloc(connect_struct, status)
....
end

void
fortran_alloc_(connect, status)
    struct gop_connect **connect;
    int             *status;
{
    *status = GOP_OK;
    *connect = (struct gop_connect *) gop_alloc_connect_structure();

    if (*connect == (struct gop_connect *) NULL)
           *status = GOP_KO;
    return;
}
```

10.4 gop_d_packet_read()

Lit un seul paquet de la Section_De_Données

Synopsis:

```
int
gop_d_packet_read(connect, buf, size, flag)
    struct gop_connect *connect
    char               *buf
    int                 size
    int                 flag
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse destination
size	taille du paquet à lire en bytes
flag	indique par GOP_TRUE que la fonction gère l'acknoledgement (cas standard).

Description:

Retourne `GOP_KO` et met `GOP_END_OF_MESSAGE` dans `gop_errno` en cas de réception d'un byte de header Fin_De_Message . Le header de Fin_De_Message reste à lire.

Si le packet est de type XDR (`connect->xdr == GOP_TRUE`), le buffer est converti à la réception.

10.5 gop_d_packet_write()

Écrit un seul paquet de la Section_De_Données

Synopsis:

```
int
gop_d_packet_write(connect, buf, size)
    struct gop_connect *connect
    char               *buf
    int                 size
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse destination
size	taille du paquet à écrire en bytes

Description:

Si le canal de communication nécessite XDR (`connect->need_xdr == GOP_TRUE`), le buffer est converti avant l'envoi.

10.6 gop_data_section_forward()

Transmet la Section_De_Données d'un message en provenance d'un canal d'entrée, sur un canal de sortie.

Synopsis:

```
int
gop_data_section_forward(from_connect, to_connect)
    struct gop_connect *from_connect
    struct gop_connect *to_connect
```

Paramètres:

from_connect	structure associée à un canal de communication en réception
to_connect	structure associée à un canal de communication en émission

Description:

`gop_data_section_forward()` se charge de la réception et de l'envoi de la Section_De_Données vers son destinataire final. La convertion en XDR est faite automatiquement selon les besoin des canaux de communications.

La transmission des packets se fait en ajustant la taille des packets émis à la taille maximum supportée par le canal de sortie (`to_connect->maxpacket`). Soit en les regroupant, soit en les explosant.

La transmission du message est arrêtée si un paquet Fin_De_Message est reçu.

Le Fin_De_Message est transmis à la place des données manquantes.

Exemple:

```
stat = gop_data_section_forward(&from_connect, &to_connect);
```

10.7 gop_data_section_read()

Lit la Section_De_Données d'un message

Synopsis:

```
int
gop_data_section_read(connect, buf, maxsize)
    struct gop_connect *connect
    char               *buf
    int                 maxsize
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse destination
maxsize	taille de buf

Description:

Lit la partie Section_De_Données jusqu'à `connect->msize` bytes ou jusqu'à la réception d'un byte de Fin_De_Message .

`gop_data_section_read()` retourne le nombre de bytes lu. Si le message a été interrompu par un Fin_De_Message , la fonction retourne le nombre de bytes lu à ce moment là en une valeur négative et pose `gop_errno` à `GOP_END_OF_MESSAGE`.

Si la valeur retournée vaut –1 (`GOP_KO`), c'est une autre d'erreur.

Dans le cas d'une réception de Fin_De_Message , seul le premier byte (le 'E') a été lu. Le header et les data du Fin_De_Message reste à lire (avec `gop_read_end_of_message()`).

10.8 gop_data_section_write()

Écrit la partie Section_De_Données d'un message

Synopsis:

```
int
gop_data_section_write(connect, buf)
    struct gop_connect *connect
    char               *buf
    int                 maxsize
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse destination

Description:

Écrit la partie Section_De_Données de taille `connect->msize` bytes.

10.9 gop_fill_bench_xdr()

Remplis une structure de type `gop_bench_xdr` avec une constante de chaque type (short, int, long, float et double).

Synopsis:

```
int
gop_fill_bench_xdr(test)
    struct gop_bench_xdr *test
```

Paramètre:

test	structure utilisée pour le test de nécessité de XDR

10.10 gop_header_fill()

Rempli le header

Synopsis:

```
int
gop_header_fill(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Remplis le header `connect->header` selon les valeur contenues dans `connect`.

La date (`connect->header.date`) est mis à jour uniquement si `header->stamp` vaut `GOP_TRUE`.

10.11 gop_first_byte_read()

Lecture du premier byte d'un packet

Synopsis:

```
int
gop_first_byte_read(connect, value)
    struct gop_connect *connect
    char               *value
```

Paramètres:

connect	structure associée au canal de communication
value	le caractère lu

Description:

Le premier caractère d'un paquet défini sa nature; 'H' pour Entête_De_Message , 'D' pour Section_De_Données , 'E' pour Fin_De_Message et 'A' pour acknowledge.

Remarque:

La lecture de paquet de type acknowledge (4bytes) n'utilise pas cette fonction.

10.12 gop_first_byte_write()

Écriture du premier byte d'un packet

Synopsis:

```
int
gop_first_byte_write(connect, value)
    struct gop_connect *connect
    char               *value
```

Paramètres:

connect	structure associée au canal de communication
value	le caractère écrit

Description:

Le premier caractère d'un paquet défini sa nature; 'H' pour Entête_De_Message , 'D' pour Section_De_Données , 'E' pour Fin_De_Message et 'A' pour acknowledge.

Remarque:

L'écriture de paquet de type acknowledge (4bytes) n'utilise pas cette fonction.

10.13 gop_h_read()

Lecture explicite du premier byte d'un header ('H')

Synopsis:

```
int
gop_h_read(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Retourne `GOP_BAD_SEQUENCE` pour toute autre valeur reçue.

10.14 gop_header_forward()

Transmet un Entête_De_Message sur le canal de communication donné. Et retourne un acknowledge vers le client selon le mode de synchronisation

Synopsis:

```
int
gop_header_forward(from_connect, to_connect)
    struct gop_connect *from_connect
    struct gop_connect *to_connect
```

Paramètre:

from_connect	structure associée à un canal de communication en réception
to_connect	structure associée à un canal de communication en émission

Description:

Le header est écrit dans `connect->header`.

La taille des paquets (`connect->header.psize`) est ajustée à la taille maximum permise sur le canal de sortie (`connect->maxpacket`)

Exemple:

```
struct gop_connect   from_connect;
struct gop_connect   *to_connect;
...
gop_header_read(&from_connect, list, &to_connect)
if (to_connect == NULL) {
   /* on est le destinataire */
   ...
} else {
   /* on est pas le destinataire */
  gop_header_forward(from_connect, to_connect);
  gop_data_section_forward(&from_connect, to_connect);
}
```

10.15 gop_header_read()

Lecture de Entête_De_Message

Synopsis:

```
int
gop_header_read(connect)
    struct gop_connect  *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

`gop_header_read()` lit l'Entête_De_Message après que le 'H' aie été lu par `gop_h_read()`. La compatibilité de version est testée et la structure `from_connect` est mise à jour selon les indications du header.

La réception d'Entête_De_Message est aquitée par un acknowledge si nécessaire (`connect->hsync == GOP_TRUE`).

10.16 gop_header_read_without_acknow()

Comme `gop_header_read()` mais sans acknowledge.

Synopsis:

```
int
gop_header_read_without_acknow(connect)
    struct gop_connect  *connect
```

Paramètre:

connect structure associée au canal de communication en réception

10.17 gop_header_write()

Envoi de Entête_De_Message

Synopsis:

```
int
gop_header_write(connect)
    struct gop_connect  *connect
```

Paramètre:

connect structure associée au canal de communication

Description:

`gop_header_write()` envoie le header (`connect->header`).

10.18 gop_io_read()

10.19 gop_io_write()

fonction de lecture/écriture bas–niveau indépendante du type de protocole.

Synopsis:

```
gop_io_read(connect, buf, size)
    struct gop_connect *connect;
    char               *buf;
    int                size;

gop_io__write(connect, buf, size)
    struct gop_connect *connect;
    char               *buf;
    int                size;
```

Paramètre:

connect	structure associée au canal de communication
buf	adresse de destination
size	taille du tableau

Description:

Ces fonctions lisent ou écrivent le nombre exact de bytes demandés.

10.20 gop_set_struct_standart()

Remplis une structure de type `gop_connect` avec des valeurs standard

Synopsis:

```
int
gop_set_struct_standart(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Variables mises à jour par cette commande:

connect−>class	Classe du message. Posé à `GOP_CLASS_COMD.`
connect−>cont	flag de continuation Posé à `GOP_FALSE.`
connect−>stamp	flag pour écrire la date dans le Entête_De_Message . Posé à `GOP_TRUE.`
connect−>hsync	Flag pour la synchronisation du Entête_De_Message . Posé à `GOP_SYNCHRO.`
connect−>dsync	Flag pour la synchronisation de la Section_De_Données . Posé à `GOP_SYNCHRO.`
connect−>stat	status du système. Posé à `GOP_STAT_OPOK`
connect−>datatype	type de données dans la Section_De_Données . Posé à `GOP_CHAR.`

10.21 gop_sig_init_handler()

Initialise un handler par défaut pour le signal ALRM et PIPE.

Synopsis:

```
int
gop_sig_init_handler()
```

Description:

La deconnection (broken pipe) doit être géree par un handler de signal pour permettre à GOP de controller se type de situation.

L'utilisation de timeout lors des opérations de lecture nécessite l'utilisation d'un handler pour la gestion du signal ALRM. Si un handler n'est pas précisé, le timeout se traduit par un arrêt du process.

Les opérations de lecture se font avec un timeout lorsque que la valeur de `connect->timeout` du canal de communication est différente de zéro. La durée de `connect->timeout` exprime un temps d'attente maximum en seconde entières.

Par exemple il définit l'attente maximum pour la réception d'un paquet lors de `gop_read*`, l'attente maximum pour un acknowledge lors de `gop_write*`.

10.22 gop_sig_handler()

Handler pour le signal ALRM initialisé par `gop_sig_init_handler()`.

Synopsis:

```
int
gop_sig_init_handler()
```

Description:

Émet un message lors de la réception du signal ALRM (voit timeout).

10.23 gop_socket_accept_connection()

10.24 gop_socket_connection()

10.25 gop_socket_init_connection()

10.26 gop_socket_close_connection()

Connections type socket

Synopsis:

```
int
gop_socket_accept_connection(connect)
    struct gop_connect *connect

int
gop_socket_connection(connect)
    struct gop_connect *connect

int
gop_socket_init_connection(connect)
    struct gop_connect *connect

int
gop_socket_close_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Ces fonctions sont les fonctions bas niveau pour des types de connections basées `socket Internet`.

Voir plus haut sous `gop_accept_connection()`, `gop_connection()`, `gop_init_connection()` et`gop_close_connection()` pour plus de détail.

10.27 gop_socket_unix_accept_connection()

10.28 gop_socket_unix_connection()

10.29 gop_socket_unix_init_connection()

10.30 gop_socket_unix_close_connection()

Connections type socket

Synopsis:

```
int
gop_socket_unix_accept_connection(connect)
    struct gop_connect *connect

int
gop_socket_unix_connection(connect)
    struct gop_connect *connect

int
gop_socket_unix_init_connection(connect)
    struct gop_connect *connect

int
gop_socket_unix_close_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Ces fonctions sont les fonctions bas niveau pour des types de connections basées `socket Unix`.

Voir plus haut sous `gop_accept_connection()`, `gop_connection()`, `gop_init_connection()` et`gop_close_connection()` pour plus de détail.

10.31 gop_socket_read()

10.32 gop_socket_write()

Fonctions bas niveau de lecture/écriture sur `socket`.

Synopsis:

```
int
gop_socket_read(connect, buf, size)
    struct gop_connect *connect
    char               *buf;
    int                 size;

int
gop_socket_write(connect, buf, size)
    struct gop_connect *connect
    char               *buf;
    int                 size;
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse de destination
size	taille du tableau

Description:

Ces fonctions lisent ou écrivent le nombre exact de bytes demandés.

10.33 gop_test_bench_xdr()

Test une structure de type `gop_bench_xdr` contenant des constante de chaque type (short, int, long, float et double).

Synopsis:

```
int
gop_test_bench_xdr(test)
    struct gop_bench_xdr *test
```

Paramètre:

test	structure utilisée pour le test de nécessité de XDR

Description:

Retourne `GOP_TRUE` si les constantes ont leurs vrais valeurs, `GOP_FALSE` sinon. La réponse de ce test indique si les valeurs passée sur un canal de communication ont besoin d'une conversion XDR.

10.34 gop_tpu_accept_connection()

10.35 gop_tpu_connection()

10.36 gop_tpu_init_connection()

10.37 gop_tpu_close_connection()

Connection type tpu

Synopsis:

```
int
gop_tpu_accept_connection(connect)
    struct gop_connect *connect

int
gop_tpu_connection(connect)
    struct gop_connect *connect

int
gop_tpu_init_connection(connect)
    struct gop_connect *connect

int
gop_tpu_close_connection(connect)
    struct gop_connect *connect
```

Paramètre:

connect	structure associée au canal de communication

Description:

Ces fonctions sont les fonctions bas niveau pour des types de connections basées `tpu`.

Voir plus haut sous `gop_accept_connection()`, `gop_connection()`, `gop_init_connection()` et gop_close_connection() pour plus de détail.

10.38 gop_tpu_read()

10.39 gop_tpu_write()

Fonctions bas niveau de lecture/écriture sur `tpu`.

Synopsis:

```
int
gop_tpu_read(connect, buf, size)
    struct gop_connect *connect
    char               *buf;
    int                 size;

int
gop_tpu_write(connect, buf, size)
    struct gop_connect *connect
    char               *buf;
    int                 size;
```

Paramètres:

connect	structure associée au canal de communication
buf	adresse de destination
size	taille du tableau

Description:

Ces fonctions lisent ou écrivent le nombre exact de bytes demandés.

10.40 gop_update_header()

Met à jour la structure associée à un canal de communication selon les paramètres contenu dans un header.

Synopsis:

```
int
gop_update_header(connect,header)
    struct gop_connect *connect
    struct gop_header  *header;
```

Paramètre:

connect	structure associée au canal de communication
header	structure type header

Variables mises à jour par cette commande:

connect–>msize	taille message en bytes.
connect–>psize	taille paquet en bytes.
connect–>mode	niveau de debug.
connect–>hsync	Flag pour la synchronisation du Entête_De_Message .
connect–>dsync	Flag pour la synchronisation de la Section_De_Données .
connect–>cont	flag de continuation
connect–>xdr	indique si la Section_De_Données est codée en XDR
connect–>datatype	type de données dans la Section_De_Données .
connect–>class	Classe du message.
connect–>stat	status du système.

Exemple:

```
struct gop_connect   *connect;
...
gop_update_header(connect, &(connect->header);)
...
```

Chapitre 11

Conseils d'utilisations

Ce chapître renseigne l'utilisateur sur les critères qui permettent de choisir le mode de synchronisation et la taille des paquets en fonction du type de communication.

11.1 Socket Internet entre Suns

- Le système gère complètement et efficacement des paquets de n'importe quelles tailles.
- D'un manière génerale plus les paquets sont gros, moins l'influence de la synchronisation se fait sentir.
- Pour les messages de taille moyenne (entre 1460 et 5000[bytes]), la synchronisation fait perdre 50% du temps de transfert.
- Pour les paquets de tailles inférieures à 1460[bytes], la non synchronisation est 10 à 25 fois plus rapide.
- Il faut absolument éviter l'envoi de paquets de tailles inférieures à 1460[Bytes] si on utilise une synchronisation. Le système perd environ 200[ms] pour l'envoi de ce type de paquet. Attention, si le dernier paquet d'un message a une taille inférieure à 1460[Bytes], on perd 200[ms].
- On peut atteindre des débits de 6[Mbits/s].

11.2 Socket Unix entre Suns

- Le système gère complètement et efficacement des paquets de n'importe quelle taille.
- Les débits sont à peu près indépendants du type de synchronisation.
- Préférer des paquets de grosses tailles.
- On peut atteindre des débits de 10[MBits/s] sur un ELC. Cette valeur dépend du type de machine.