Sujets de Projet 2016

Synchronisation des Processus Posix

Gestion des impressions dans un système

VAYSSADE Jehan-Antoine

Binôme Solitaire, Groupe IGAI

http://sleek-think.ovh/websvn/listing.php?repname=cups

svn://sleek-think.ovh/cups/

Written by VAYSSADE Jehan-Antoine <javayss@sleek-think.ovh>, December 2016

I. Les machines sites

Chaque machine est représentée par une application dédiée →

- server → qui représente évidemment le Spooler d'impression

- client → qui représente évidement le deamon utilisateur d'interconnexion avec le Spooler
  - idéalement, il faudrait transmettre l'uid du client pour restaurer la queue d'impression
  - si un utilisateur se déconnecte, son processus sur le serveur est alors suspendu, car il est non joignable pour recevoir une notification ou une demande de document.

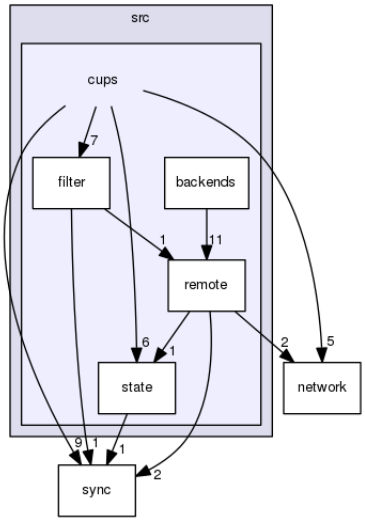
- printer → qui représente une imprimante distante dans un protocole (IPP, TALK, JET, ...)
  - n'intervient pas dans le modèle de synchronisation puisque le réseau garantit la synchronisation via une socket unique, les commandes sont simplement transmises via une imprimante virtuelle (par datagramme) qui utilise simplement le model Printer « fourni » par Jehan (d'où la présence des copyright).
  - et de plus les imprimantes distantes sont censés déjà exister ;)

Chaque machine a évidemment une correspondance au sein du modèle objet du serveur à la fois pour le réseau et pour les différentes synchronisations →

- server → Spooler (net)
  - SpoolerQueu → stream<PrinterJobQuery>
  - SpoolerDispatch → consumer<PrinterJobQuery> to producer<Job*>

- client → User (net)
  - remote::RemoteDocument (file transfert)
  - UserJobs → producer<PrinterJobQuery>

- printer → remote::RemotePrinter et remote::PrinterServer(net)
  - Printer consumer<Job*>
  - PrinterQueu stream<Job*>
  - PrinterJobs producer<Job*>



(Interaction des modules)

Plusieurs classes et classes génériques ont été développées afin de simplifier l'utilisation dans le framework principal, que l'on peut retrouver un peu partout soit en héritage, soit en attribut, ou encore en fonction objet.

Tout ce qui concerne le réseau (adress, file_descriptor, socket, …) et ce qui concerne la synchronisation via un mpmc (produceur, stream, consumer) la gestion des processus (process et lock_guard) est donc développé comme module externe.

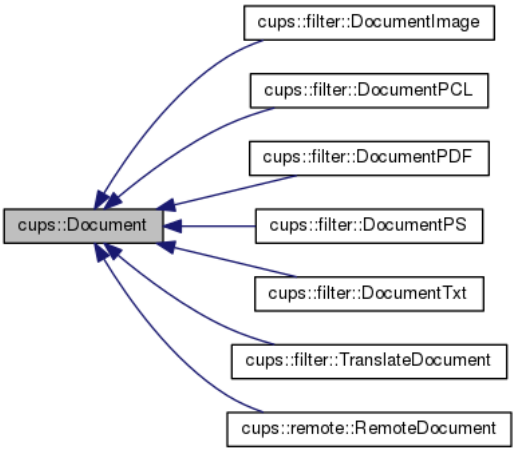
Le reste de l'application est également fragmenté en sous-module pour simplifier la conception et diminuer le couplage.

2. Les utilisateurs du service d'impression

Comme demandé l'utilisateur peut imprimer différents types de fichiers, la machine utilisateur fait une pré-conversion vers un format unique pour le spooler (CUPS-PostScript), qui est ensuite transformé vers un format spécifique à l'imprimante dans un thread avant d'être traité.

Lorsque la requête d'impression est consommée pour être dispatchée vers l'imprimante demandée, le Document est alors « décorée » par un TranslateDocument.

Ce TranslateDocument se charge de lancer un thread pour récupérer localement le document, via RemoteDocument et enregistrer dans un fichier temporaire afin d'appliquer un traitement propre à l'imprimante vers un autre fichier temporaire unique qui correspond à la liste de commandes à utiliser.



Dans le diagramme si contre, un Document est chargé coté client, et ajouter en file d'attente pour traitement de conversion local et fait office de pré-filtre (conversion vers cups-postscript) qui est alors reçu et traité sur le serveur en post-filtre (conversion vers cups-raster→ TranslateDocument).

Lorsque le Job associé est pris en charge par l'imprimante, cette dernière attends simplement que le processus de conversion arrive à son terme avant de lancer le traitement.

Cette modélisation permet en plus que chaque utilisateur puisse définir ses méthodes de conversion ou en installer d'autres localement(utilisation du design pattern strategy).

Pour simplifier l'utilisation, et comme « conseillé » un DocumentFactory est disponible afin de fournir le type de Document correspondant. La dernière classe remote::RemoteDocument est là pour s'occuper du transfert de fichier entre l'utilisateur et l'imprimante.

L'utilisateur peut évidemment envoyer les commandes suivantes au serveur :

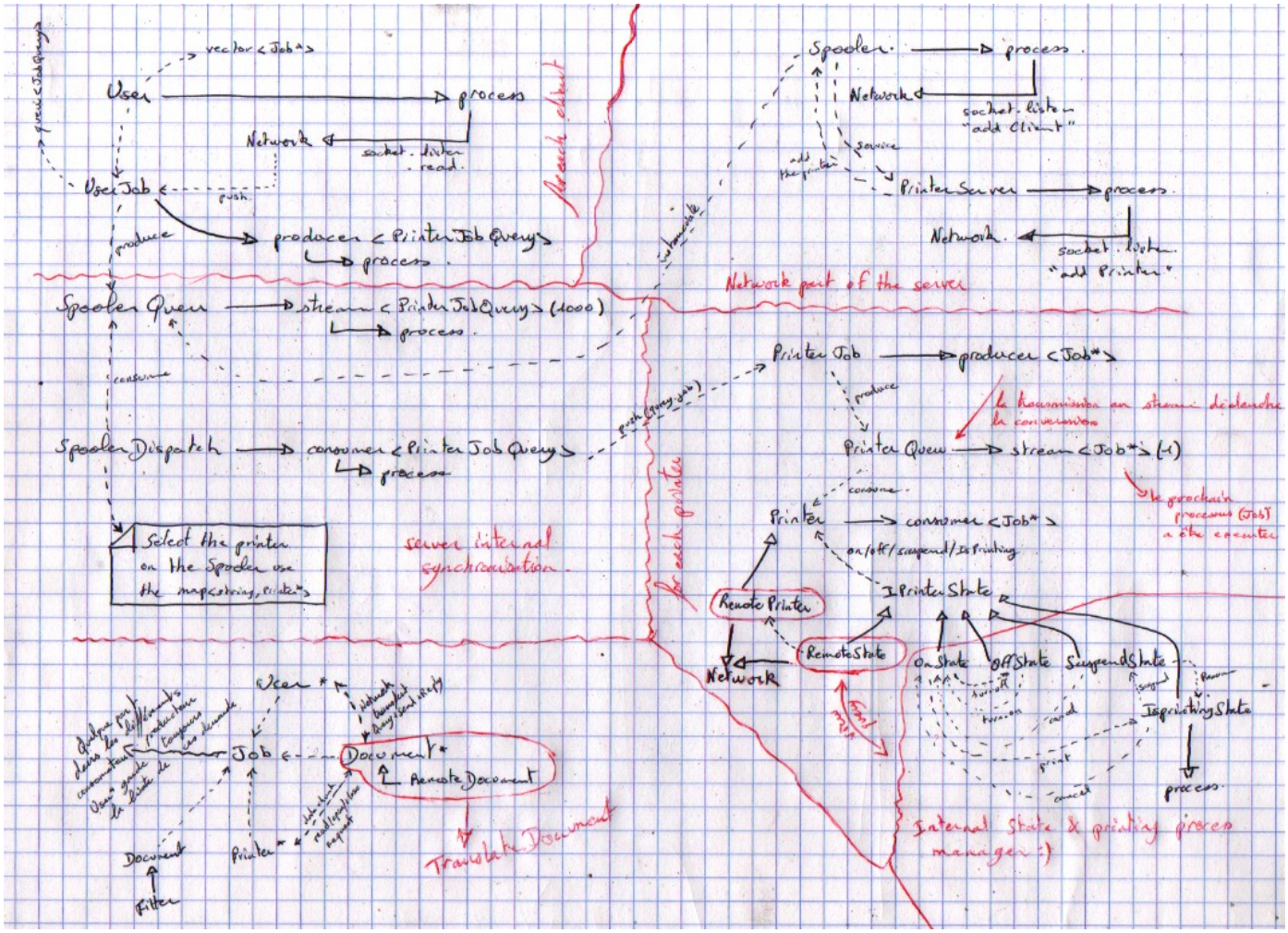
- soumettre des travaux d'impression par le réseau via l'application cliente
  - avec une imprimante spécifique (via son pseudo url)
  - définir un nombre de copies
  - recto / verso
  - l'identifiant du client

- connaître l'état d'avancement d'un travail d'impression
  - le serveur garde la liste des travaux soumis
  - à chaque fois que le Job change d'état sur le Spooler le client est notifié (packet)

- changer l'état d'un travail d'impression
  - si il est en attente, il peut être annulé
  - si il est en cours d'impression, il peut être soit suspendu, soit annulé
  - si il est suspendu, il peut être soit repris, soit annulé

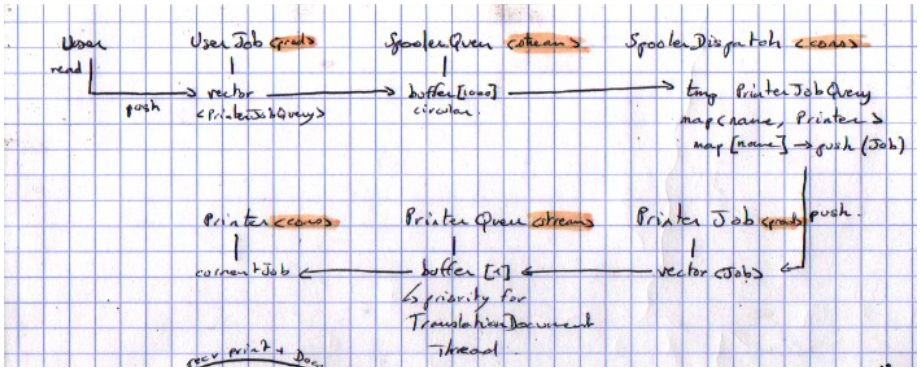
III. Le serveur d'impression

Un bon dessin vaut mieux qu'un roman, le diagramme suivant représente simplement les différentes interactions entre les objets sur le serveur. L'architecture complet se trouve en annexe



Lorsqu'un client se connecte au serveur, deux threads sont créés, un pour le réseau (User), et l'autre (UserJob) pour produire un PrinterJobQuery qui contient le nom de l'imprimante et le Job, qui est alors stocké à la fois chez le client (liste de Job soumit pour mise à jour) et sur le stream SpoolerQueu.

Un premier consommateur SpoolerDispatch est présent pour distribuer la requête depuis SpoolerQueu vers le bon producteur PrinterJob (qui stocke la liste d'attente pour imprimante), ce dernier est attaché a un PrinterQueu de taille 1 qui permet de protéger la transmission de la prochaine tache que le consommateur (l'imprimante) va prendre en charge.



On a donc :

- 2 thread client (socket+UserJob)
- 3 thread serveur (socket+SpoolerQueu+SpoolerDispatch)
- 2 thread printer (PrinterJob+Printer)
- 1 thread par TranslateDocument
- 1 thread PrinterServer (dynamicaly register new Printer)
  - instantiate RemotePrinter & related thread

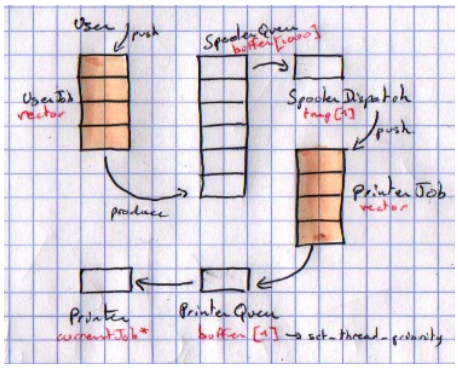
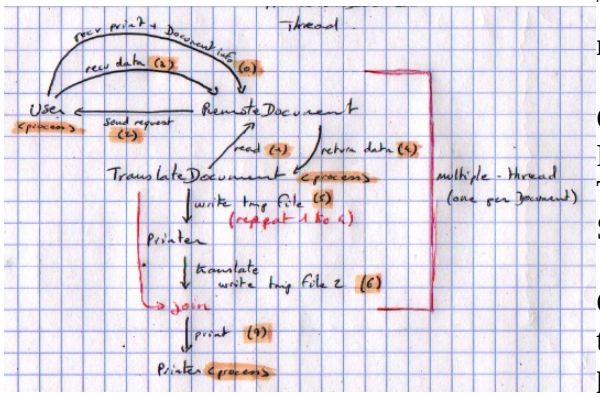


Illustration 1: request to print

Pour la transmission et la traduction (CUPS-Filter et CUPS-Raster) :

Lorsque le client fait une demande d'impression, les informations du document sont envoyées.



Tel que le chemin sur la machine distante, le type, le nombre de lignes, et le nombre de pages.

Ces informations sont alors stockés dans un objet RemoteDocument qui sera alors décoré par un TranslateDocument lors du passage entre SpoolerDispatch et PrinterJob.

Ce TranslateDocument est en charge de lancer le thread qui va permettre de faire la demande de chaque partie du fichier (etape 1 a 4 sur le diagramme)

L'ensemble de ces informations est alors stocké dans un fichier temporaire dans /tmp. Une fois que le fichier a été complètement reçu, TranslateDocument va utiliser la méthode translate (CUPS-Raster) de l'imprimante utiliser par le Job.

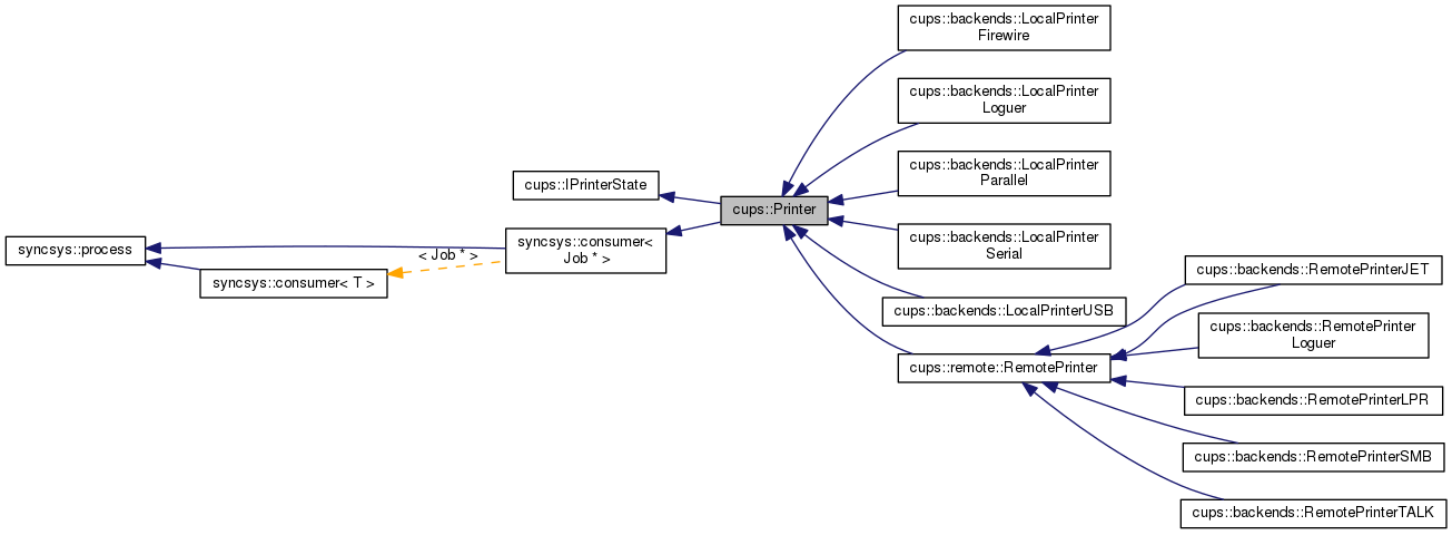
Cette méthode va s'occuper de traduire le fichier distant (envoyer en format unique PostScript donc pré-filtré) vers le format spécifique de l'imprimante cible enregistrée encore une fois dans un nouveau fichier temporaire.

Chaque type d'imprimante, ré-implémente donc la méthode translate, pour faire la traduction vers le format qui lui correspond, cette méthode ne modifie pas l'objet instancié mais ne peut pas être déclaré statique pour bénéficier des tables-virtuelles.
Elle est donc simplement const elle peut donc être appelée simplement depuis n'importe quel thread, sans aucune synchronisation.

A noter que ce modèle a été conçu pour être futurement modifiée de tel-sorte que le Spooler va finalement se connecter à un mini serveur pour chaque imprimante pour qu'il soit possible d'avoir plusieurs serveurs d'impression qui ont différentes imprimantes partagées, permettant de gérer le droit ou de répartir la charge.

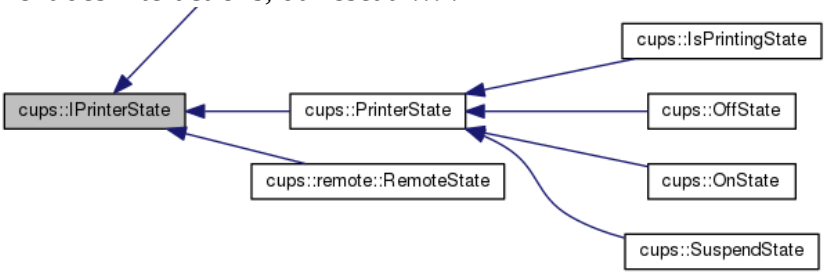
IV. Les imprimantes

Comme demandé une imprimante peut être locale ou distante, j'ai donc simplement modélisé une classe Printer qui est dérivée comme suit :



Chaque sous-classe est donc la représentation d'un protocole de communication avec une imprimante locale (USB/Firewire/Parallel/Serial/Loguer) ou distante via RemotePrinter et les sous-protocoles (JET/LPR/SMB/TALK/Loguer).

PS : actuellement les classes *Loguer sont ici pour symboliser les Imprimantes en mode debug, donc enregistrement des interactions, du réseau … .



Chaque imprimante dispose aussi d'une vue sur l'état dans lequel il se trouve et permet de gérer le processus d'impression, c'est donc un peu comme une machine à état : chacun donne la main à l'autre et permet d'intéragir indirectement sur le processus géré par IsPrintingState (attente, reprise, impression, annulation).

Lorsque l'état courant reçoit le changement d'état suivant :

- print : le processus est lancé (si disponible)
- suspend : le processus est suspendu avec un sémaphore (si en impression)
- résume : post sur le sémaphore précédemment décrit (si suspendu)
- cancel : cancel le thread d'impression (si non disponible)

La class Printer implémente aussi la « méthode » std::string translate (const std::string &filename)

qui une fois surchargée traduit le CUPS-PostScript en CUPS-Raster sur la documentation cups. Pour ce TP cela représente donc les filtres associées à une imprimante, puisque j'ai décidé d'appliquer un pré-filtre que le client va effectuer vers un format standard (ici PostScript) conformément a CUPS et effectuer sur un processus pour traitement parallèle.

V. Travail demandé

Le makefile fourni dispose des options suivantes :

- all, debug, clean et dir (créer les dossiers de compilation pour les *.o)

Si l'exposé de ce compte rendu n'a pas été suffisant pour la compréhension de cette application, la documentation html peut être générée avec Doxygen, l'ensemble des classes et des diagrammes de collaborations seront visibles, de plus le code source se suffit à lui même pour documentation .

« any code should be self-documenting with a clear semantic »

La version démo est celle générée par le makefile et est la pour permettre le test.
Le script « stress-test.sh » est présent pour tester la forte influence de demande d'impression.

Retour d'expérience :

Toutefois, la finalité de ce projet n'est pas d'alignée des lignes de code C, mais plutôt de bien concevoir un système parallèle, voir réparti, et d'expliciter la synchronisation et la communication mises en œuvre pour permettre aux activités qui le composent, d'assurer correctement le service demandé.

Ah bon ???

Parce que la partie réseau est vraiment nécessaire à la compréhension du multi-threading ? de même pour les filtres ? Et le transfert de document ? Et des utilisateurs ? Et … ? tout cela se ramène a un « MPMC queue » et quelques sémaphores d'événements afin d'injecter ou d'attendre une information dans un autre thread.

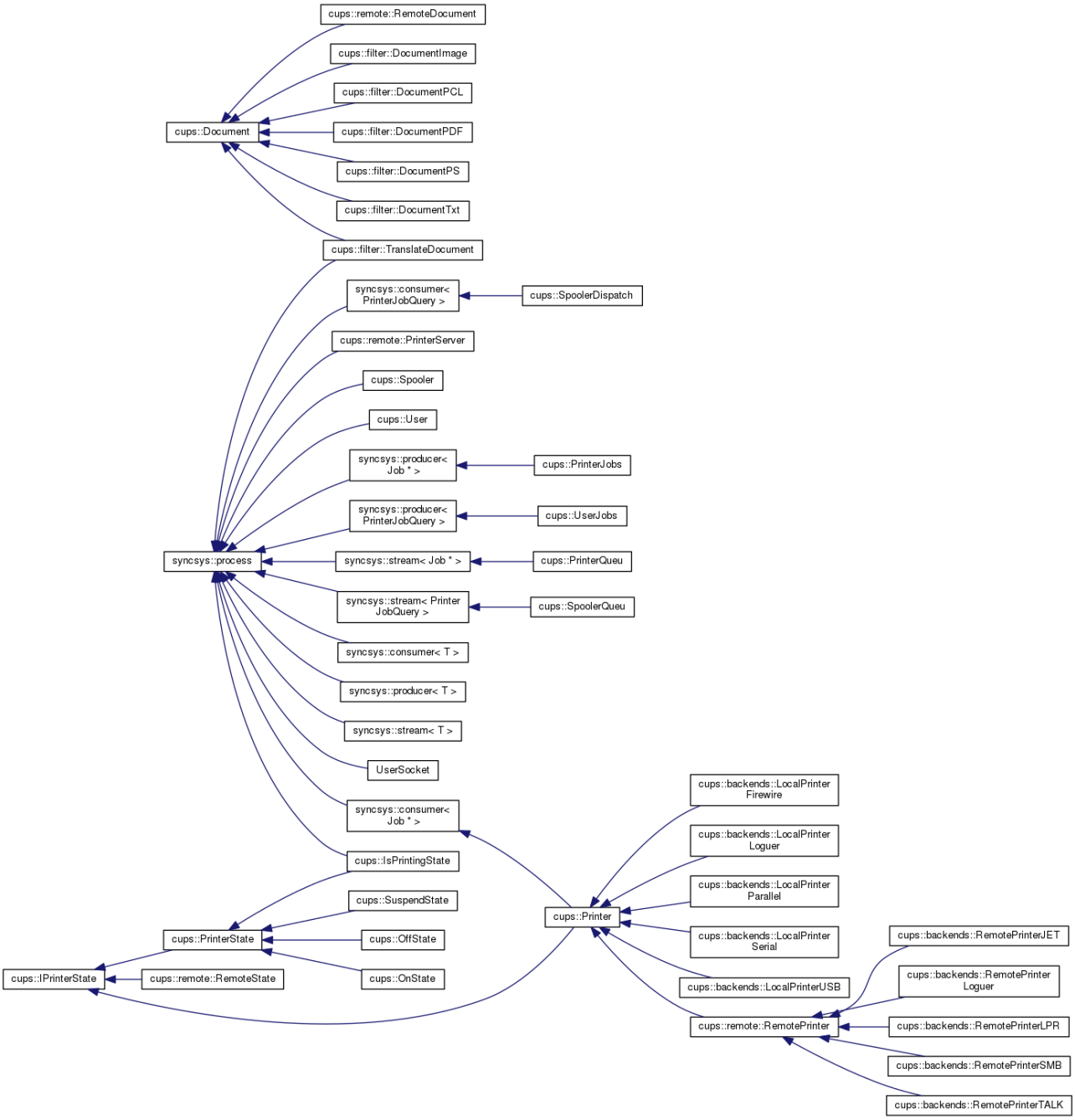
De surcroît la « parti » réseau donnée, en plus d'être absolument immonde, est synchrone ! ce qui signifie que les données ne sont pas reçus en parallèle → safe

Donc retour d'expérience nul, puisque :

1. L'ensemble processus/consumer/stream/producer avais été fait plusieurs fois
    1. TP1 : template<typename T, int size, int shared = 0> class buffered_rwlock ;
    2. TP1 bis : problème du pont : class process/stream/producer/consumer ;
    3. TP2 : producteur, consommateur
    4. TP2 bis : problème du pont revu (producteur, consommateur)
2. le but de ce TP « projet » n'est donc visiblement pas la synchronisation (déjà fait)
    1. le temps passé sur la synchronisation est donc trivial comparé au reste ~17 %
    2. contrairement au réseau inutile dans la simulation ~20 %
    3. la conception ~45 %
    4. le rapport ~15 %
    5. et de la recherche d'information

De plus cela n'a pas été spécialement enrichissant pour les autres domaines. Ce projet aurait pu être bien s'il avait été partagé entre plusieurs matières mais ici, comme ce n'est que pour le de la synchronisation, il semble que c'est beaucoup de travail gratuit/fournis pour pas grand-chose ! merci d'économiser le temps de vos élèves !

VI. Annexe


cups::remote::RemoteDocument
cups::filter::DocumentImage
cups::filter::DocumentPCL
cups::Document
cups::filter::DocumentPDF
cups::filter::DocumentPS
cups::filter::DocumentTxt
cups::filter::TranslateDocument
syncsys::consumer< PrinterJobQuery >
cups::SpoolerDispatch
cups::remote::PrinterServer
cups::Spooler
cups::User
syncsys::producer< Job * >
cups::PrinterJobs
syncsys::producer< PrinterJobQuery >
cups::UserJobs
syncsys::process
syncsys::stream< Job * >
cups::PrinterQueu
syncsys::stream< Printer JobQuery >
cups::SpoolerQueu
syncsys::consumer< T >
syncsys::producer< T >
syncsys::stream< T >
UserSocket
syncsys::consumer< Job * >
cups::IsPrintingState
cups::SuspendState
cups::PrinterState
cups::OffState
cups::IPrinterState
cups::remote::RemoteState
cups::OnState
cups::backends::LocalPrinter Firewire
cups::backends::LocalPrinter Loguer
cups::backends::LocalPrinter Parallel
cups::Printer
cups::backends::LocalPrinter Serial
cups::backends::RemotePrinterJET
cups::backends::LocalPrinterUSB
cups::backends::RemotePrinter Loguer
cups::remote::RemotePrinter
cups::backends::RemotePrinterLPR
cups::backends::RemotePrinterSMB
cups::backends::RemotePrinterTALK