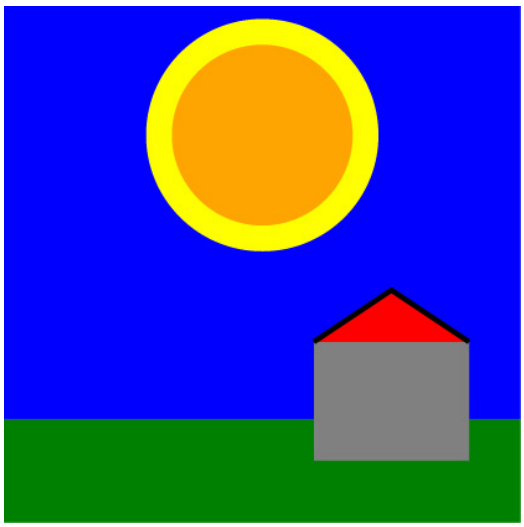
Rendu SVG piloté par le réseau

Romain Corbeau
Ionas Neonakis
Samir Toularhmine
Anaël Trimoulet



SOMMAIRE

DESCRIPTION DU PROJET

Répartition des tâches

Afin de réaliser le projet, nous nous sommes répartis les tâches selon deux axes principaux:
- Le client, géré par Romain et Anaël
- Le serveur, géré par Samir et Ionas
Une fois ces deux axes principaux établis, nous avons choisi chacun une partie plus précise à faire:
- Pour Samir: La fenêtre d'affichage GTK et la gestion du SVG
- Pour Ionas: La gestion du réseau et des données (côté serveur)
- Pour Romain: La mise en place du réseau et des données (côté client)
- Pour Anaël: La modification des données et l'interaction utilisateur

Cependant, en fonction des points forts de chacun, la réalité fut quelque peu différente:
- Samir a réalisé sans soucis ses tâches et a également intégré un client graphique en se basant sur les fonctions du client console. Il s'est également occupé de la structure globale du projet et de la création du Makefile.
- Ionas a réalisé une classe (Dataparser) pour gérer efficacement les données de la partie partie client et serveur ainsi que l'implémentation du réseau côté serveur en se basant sur les recherches de Romain.
- Romain a cherché comment réaliser un échange simple de données en UDP en minimisant la charge réseau et a implémenté cela côté client en plus de réaliser la modification des données et l'interaction utilisateur.

Pour ce qui est du rapport, nous avons chacun écrit notre partie et tout rassemblé dans ce même document.

Client - La console

Parallèlement au serveur, le client a été développé avec peu de structuration, en commençant par l'envoi de messages au format UDP puis en intégrant une partie interaction avec l'utilisateur.

Afin de réaliser avec succès l'envoi de datagramme UDP, nous avons suivie différents sites web (forum, documentation, cours (en particulier celui-ci https://bousk.developpez.com/cours/reseau-c++/UDP/01-introduction-premiers-pas/)) puis vérifier à l'aide du logiciel WireShark si les messages envoyées correspondaient bien à des datagrammes UDP conforme.
Ensuite, nous avons intégré l'interaction avec le client qui, à l'aide d'un système de boucles imbriquées, itère sur les éléments pour les afficher en fonction des choix de l'utilisateur.

A noter qu'au début, le client envoi au serveur une map contenant un point d'interrogation en clef et en valeur pour récupérer les informations modifiables.
Cependant, le client était très mal structuré (tout dans la même classe, des fonctions compliqué, etc). C'est pourquoi, nous avons décidé de réorganiser tout le code selon une architecture MVC très simplifiée (qu'on appellera architecture MV) contenant un modèle (les fonctions essentielles) et deux vues (une pour l'affichage console et une pour l'affichage graphique).
Pour plus d'information sur l'implémentation du client, se reporter à «Choix d'implémentation - Le client en général)

DESCRIPTION DU PROJET

Client - L'interface graphique

Afin de proposer une interface un peu plus agréable d'utilisation, nous avons mis au point une interface graphique. Celle-ci a été effectuée avec GTK 3 et communique grâce à une architecture Modèle – Vue avec le modèle client.

Les fonctionnalités du client graphique sont similaires à celles du client console, avec l'ajout de la suppression d'un attribut en plus.

Lors de l'élaboration de ce client, nous n'avons pas rencontré de problème particulier.

Serveur - Les données

Pour la partie serveur nous avons créé un serveur en UDP qui possède l'adresse IP suivante : 127.0.0.1 et qui écoute sur le port 6789.
Il contient l'image SVG qu'il va être amené à modifier et est constamment en attente de réceptions de données.
Dès qu'un paquet est reçu, le buffer est converti en cbor::binary, soit du cbor encodé.
Une fois le buffer traduit, le serveur fait appelle à la fonction lireMessage du Dataparser qui transforme ce message encodé un cbor::map puis ensuite en en vector<Message>. Message étant un objet contenant un nom et une valeur.
Le serveur va ensuite vérifier si le message reçu est un point interrogation '?'.
Le cas échéant, le serveur envoie sous forme de string tous les attributs driven de l'image au client, suivi de tous les attributs des attributs style de l'image. Ainsi côté client il n'y aura plus qu'à afficher ces attributs
Sinon, le serveur fait appel à la fonction update de Window afin de modifier les valeurs des attributs reçus.

DESCRIPTION DU PROJET

Serveur - L'affichage

Afin de pouvoir afficher une image SVG rendue dynamiquement à l'écran, nous avons utilisé plusieurs librairies :
→ GTK 3 pour le gestionnaire de fenêtre qui affichera le canva Cairo
→ Cairo pour avoir un canva sur lequel afficher l'image SVG
→ Librsvg pour pouvoir traduire les informations du fichier SVG en données compréhensibles par Cairo et pour modifier dynamiquement en mémoire ce fichier
→ TinyXml2 pour parser efficacement le fichier SVG.
→ TinyExpr afin de pouvoir envoyer des expressions évaluables
→ Color pour nous aider sur la conversion de couleur RGB en HSV et vice-versa

Nous avons une classe Window qui encapsule toute la logique de l'affichage graphique à l'écran. En effet, elle encapsule des informations comme la taille ou le titre de la fenêtre mais surtout : une fenêtre GTK, un handler pour le fichier SVG, et un object TinyXml2::XMLDocument qui sera modifié à chaque nouveau message reçu par le serveur.

Voici donc la logique adoptée par la fenêtre d'affichage lors de la réception d'un message (en sachant que ce dernier peut contenir plusieurs attributs à modifier) :

1 : Réception du message par le serveur
2 : Transmission à la fenêtre d'affichage
3 : Parsing de chaque valeur reçue (évaluation de l'expression en valeur calculée, application de la Regex)
4 : Modification du fichier XMLDocument
5 : Rafraîchissement de la fenêtre GTK .

FONCTIONNEMENT

Manuel d'utilisation

Lancement du programme

Compilation du projet (se placer à la racine du projet):
make

Suite à la compilation, deux exécutables sont créés :
- start_serveur qui va s'occuper de lancer le moteur de rendu SVG
- start_client qui va lancer l'interface client pour interagir avec le serveur

Pour pouvoir lancer le serveur, il faut lui préciser le nom d'un fichier SVG présent dans le dossier serveur/resources.
Il ne faut pas préciser le chemin vers le fichier SVG, seulement son nom avec son extension exemple :
./start_serveur maison.svg

Note : Il faut impérativement que le fichier SVG soit présent dans le dossier resources.

Pour pouvoir lancer le client, il faut lui préciser un argument d'activation de l'interface graphique.

Lancer le client avec l'interface graphique : ./start_client 1
Lancer le client sans l'interface graphique : ./start_client 0

Utilisation du client (console)

Une fois le client lancé, le message suivant apparaît dans la console: Message envoyé.
Il sera suivi (après quelques millisecondes) d'une liste d'éléments dit «driven» qui correspondent aux items modifiables (Liste page 7). Il suffit d'écrire exactement le nom de l'un d'eux afin de le modifier.
Les éléments se terminant par «_style» représentent les règles CSS de cet objet. Par exemple, «sun_style» permet de modifier les règles CSS de l'élément sun. Ces items sont appelés «éléments de style». (Liste page 7)

Si on choisi un élément driven, le programme demandera la nouvelle valeur à affecter à l'item.
Si on choisi un élément de style, le programme listera toutes les règles CSS modifiables de l'objet.
Il faudra ensuite en choisir une parmi celles-ci puis modifier sa valeur comme si c'était un élément driven.
Si la règle CSS a déjà eu une valeur affectée dans la même session du programme, elle sera écrasée.
Dans tous les cas, si la valeur envoyée n'est pas correcte, la modification sera ignorée par le serveur.

Après chaque modifications, le programme demande si il y en a d'autres à apporter à l'image. On peut alors répondre «yes» «y» «oui» «o» pour oui ou «no» «n» «non» pour non.
Si on répond oui, le programme retourne au début et on peut à nouveau modifier un élément.
Sinon, il envoi les données au serveur et s'arrête.

FONCTIONNEMENT

Utilisation du client (interface graphique)

Après avoir lancé l'exécutable, une fenêtre s'affiche. Une liste déroulante est alors proposée à l'utilisateur afin de choisir l'attribut qu'il veut modifier.
Une fois cet attribut choisi, il entre une valeur correspondante puis clique sur Ajouter.
L'utilisateur peut ajouter/supprimer (bouton Supprimer ou touche du clavier Suppr) autant d'attribut qu'il le souhaite. Une fois prêt, il peut cliquer sur Envoyer pour notifier le serveur des changements.

Liste des éléments driven

Nom	**Description**	**Valeur attendue**
sun_x	Modifie la position du soleil sur l'axe horizontale	Nombre entier
sun_style	Élément de style modifiant le soleil	
sun_y	Modifie la position du soleil sur l'axe vertical	Nombre entier
sun_fill	Modifie la couleur du soleil	Couleur (en hexadécimal ou selon le code HTML)
sun_opacity	Modifie l'opacité du soleil	Nombre flottant entre 0 et 1

Liste des éléments de style

Élément driven	**Élément de style**	**Valeur attendue**
sun_style	margin	Nombre entier
	padding	Nombre entier

FONCTIONNEMENT

Client - Structure globale et problèmes rencontrées

Comme évoqué précédemment, le client actuel suit une architecture MV très simplifiée avec un modèle contenant des fonctions pour les vues.
Étant donnée l'absence de contrôleur, il est nécessaire d'instancier directement la vue choisie qui elle même instanciera le modèle.
Le modèle se résume à la classe «functions» et possède:
En terme de variables:
- Une socket pour le réseau
- Une map (cbor) qui contient les données à envoyer au serveur
- Un vecteur qui contient la liste des éléments driven
- Une map contenant en clef les éléments driven et en valeur un vecteur pour la liste des éléments de style

En terme de fonctions:
- Un constructeur initialisant tout ce qui est nécessaire pour le réseau
- Un destructeur
- Une fonction «sendData» qui permet d'envoyer des données au serveur
- Une fonction «insertInData» qui permet d'ajouter un élément dans la map contenant les données à envoyer
- Quelques fonctions utilitaires pour analyser des éléments.
- Deux fonctions renvoyant les éléments driven ou les éléments de style pour un élément driven donné.

Les vues implémentent une interface iClient qui contient la fonction «start». Cette dernière est automatiquement appelé au lancement.

Pour le client en console, la structure est assez simple:
- On vérifie que le modèle contient bien des données
- On affiche les éléments driven (showDrivenItems)
- On sélectionne celui à modifier (SelectItemToModify)
- On le modifie (modifyItem, appelé 2 fois si on modifie un élément de style)
- On choisi si recommence pour affecter une nouvelle règle ou si on quitte le programme (finish)

FONCTIONNEMENT

Serveur - Structure globale et problèmes rencontrées

Un problème assez vite rencontré a été que la fenêtre GTK ait sa propre boucle infinie encapsulée et que le serveur ait la sienne. Il nous a donc fallu lancer la fenêtre principale dans un second thread afin que le serveur puisse appeler les méthodes de la fenêtre sans blocage.

Également, nous avons fait face à un manque de fonctionnalité de parsing dans TinyXml2. En effet, il n'y a pas de fonction pour vérifier si la valeur de l'élément est valide ou non. Cela étant, nous avons donc créé un système de typage pour nos attributs. Pour ce faire, nous avons mis au point un fichier types.cfg (dans le dossier config) qui associe à chaque type d'attribut SVG une regex correspondante. Celle-ci est basée sur la grammaire du type des attributs que nous avons pu trouver en nous documentant sur le format SVG. Ce système nous oblige donc, pour chaque élément driven, de passer un type d'attribut en attribut. Grâce à cela, lors du parsing de la valeur, nous stockons dans une map de signature <TypeAttribut, Regex> tous les types trouvés dans le fichier types.cfg et si la Regex s'applique alors le message est traité. De plus, ce typage nous permet d'effectuer une interpolation de valeur appropriée dont nous parlerons plus loin.

L'évaluation de la valeur est réalisée juste avant le parsing, grâce à la librairie TinyExpr. Cela nous permet de comprendre un grand nombre d'expression logique et mathématique.

Après le parsing, il faut modifier les données stockées en mémoire du fichier SVG. A la suite d'une première réflexion nous avions conclu qu'il nous fallait modifier le fichier SVG directement, enregistrer par dessus à chaque modification puis recharger le fichier à chaque message traité.
Seulement, en faisant ceci, nous ne faisions pas usage des données stockées en mémoire du fichier SVG. Nous avons donc décidé de modifier directement cet objet et ne pas recharger le fichier à chaque modification : c'est donc beaucoup plus efficace.

La modification de la valeur peut s'effectuer avec une transition, plus ou moins fluide en fonction de la valeur l'attribut delay précisé dans la balise driven.
Cette interpolation s'applique sur tous les types disponible grâce à une map de signature <TypeAttribut, FonctionInterpolation> qui associe à chaque type d'attribut une fonction d'interpolation qui va générer un vecteur de valeurs comprises entre l'ancienne et la nouvelle.

FONCTIONNEMENT

Serveur - Structure globale et problèmes rencontrées

Comment est effectuée l'interpolation de valeur?

Au vu de la grande variété de type d'attribut nous avons conclu qu'il fallait une fonction d'interpolation pour chaque type d'attribut.

Celle-ci va se baser sur une formule qui, pour un temps T donné, va nous donner une valeur à insérer dans le vecteur de valeur :

$$ancienneValeur + ((nouvelleValeur - ancienneValeur) * t)$$

Grâce à cette formule, nous avons pu interpoler n'importe quelle valeur numérique.

Et pour les couleurs ?

Pour les attributs de couleur, nous avons suivre un processus d'interpolation qui a demandé une grande réflexion afin que cela soit visuellement beau à l'écran.
Le problème est qu'une couleur est donnée sous forme de «color-keyword » par l'utilisateur.
La première chose à faire est de transformer ce mot en une valeur hexadécimale, et nous avons pu faire ceci grâce à une map de signature <Color-Keyword, ValeurHexadécimale> qui associe une valeur hexadécimale à chaque color-keyword. Cette map est initialisée grâce à une fichier colors.cfg : il est donc facile de rajouter une couleur compatible (à noter qu'il faut également rajouter le color-keyword dans la regex concernée dans le fichier types.cfg).

Ensuite, nous avions simplement converti les valeurs hexadécimal en composantes RGB, puis interpolé chaque composante. Cela fonctionnait, seulement, l'oeil ne fonctionnant pas avec un système RGB, nous obtenions un rendu assez grisâtre.

Pour palier à ce problème, il nous a fallu utiliser le système de couleur HSV (Hue Saturation Value).
Ce système ne se base pas sur l'intensité des composantes rouge, verte et bleue mais va se baser sur une palette de couleur circulaire dont on précisera la position du pointeur de couleur et son intensité.

Après avoir passé les couleurs en RGB, puis en HSV, nous avons ensuite pu interpoler les valeurs, puis repasser les valeurs obtenues en RGB, puis en hexadécimal, format de couleur compréhensible par librsvg.

Note : cette interpolation ne s'applique pas sur les attributs style.

Enfin, après modification de l'objet SVG en mémoire, il nous a fallu rafraîchir la fenêtre manuellement. En effet, GTK rafraîchit la fenêtre uniquement lors d'un événement de redimensionnement, focus ou déplacement. Afin de rendre la modification visible de manière instantanée, nous avons donc fait usage de la fonction gtk_widget_queue_draw() nous permettant ce rafraîchissement.

CHOIX D'IMPLÉMENTATION

Client - En général

Après avoir codé le client sans architecture prédéfinie, la réutilisation du code pour différentes vues était compliquée, c'est pourquoi nous avons choisi de réorganiser le code et de partir sur une architecture MV.
Cependant dans l'état actuel, il n'est pas utile de lancer plusieurs vues en même temps sur le même programme, l'intégration d'un contrôleur entre le modèle et les vues semblait superflu, c'est pourquoi nous avons décidé de ne pas en mettre.

De plus, il aurait été possible d'envoyer plusieurs messages au serveur dans le même programme, comme par exemple, à chaque modifications au lieu de toutes les attendre.
Nous n'avons pas privilégié cette voie au profit de l'actuelle qui regroupe toutes les modifications en un seul datagramme UDP afin de minimiser la charge réseau.

Par ailleurs, nous nous sommes également posé la question sur le fait d'envoyer plusieurs fois la même règle CSS dans le même message mais avec des valeurs différentes. Par exemple: Envoyer pour sun_style: fill: green puis fill: red.
Nous avons choisi d'écraser l'ancienne valeur au profit de la nouvelle mais il aurait également été possible de faire en sorte que le soleil passe d'abord par le vert avant de devenir rouge.

Client - Gestion des données

Pour la partie client nous avons d'abord créé un client_console qui interagit avec l'utilisateur via la console. Ce client, dès la création envoie un paquet contenant une cbor:map avec ces valeurs : {{'?','?'}} afin que le serveur puisse envoyer les attributs de l'image qui sont modifiables. Ensuite sur la console du client on y voit y apparaître les éléments driven modifiables. L'utilisateur choisit les éléments qu'il souhaite modifier avec les nouvelles valeurs. Une fois fini le client transforme le message de l'utilisateur en cbor::map puis en cbor::binary et envoie ces données via un paquet UDP, puis le client s'arrête.

Serveur - Réseau

Pour l'implémentation réseau nous avons décidé d'utiliser un serveur UDP qui n'est donc pas connecté. Pour l'envoi des données nous avons utilisé des objets cbor de la librairie cbor11 car elle paraît plus simple d'utilisation que la librairie lib_cbor. la librairie cbor11 propose de passer d'un std::map à une cbor::map instantanément puis en objet cbor qui lui peut être sérialisé ou «encodé» en faisant appel à la fonction encode de cette librairie.