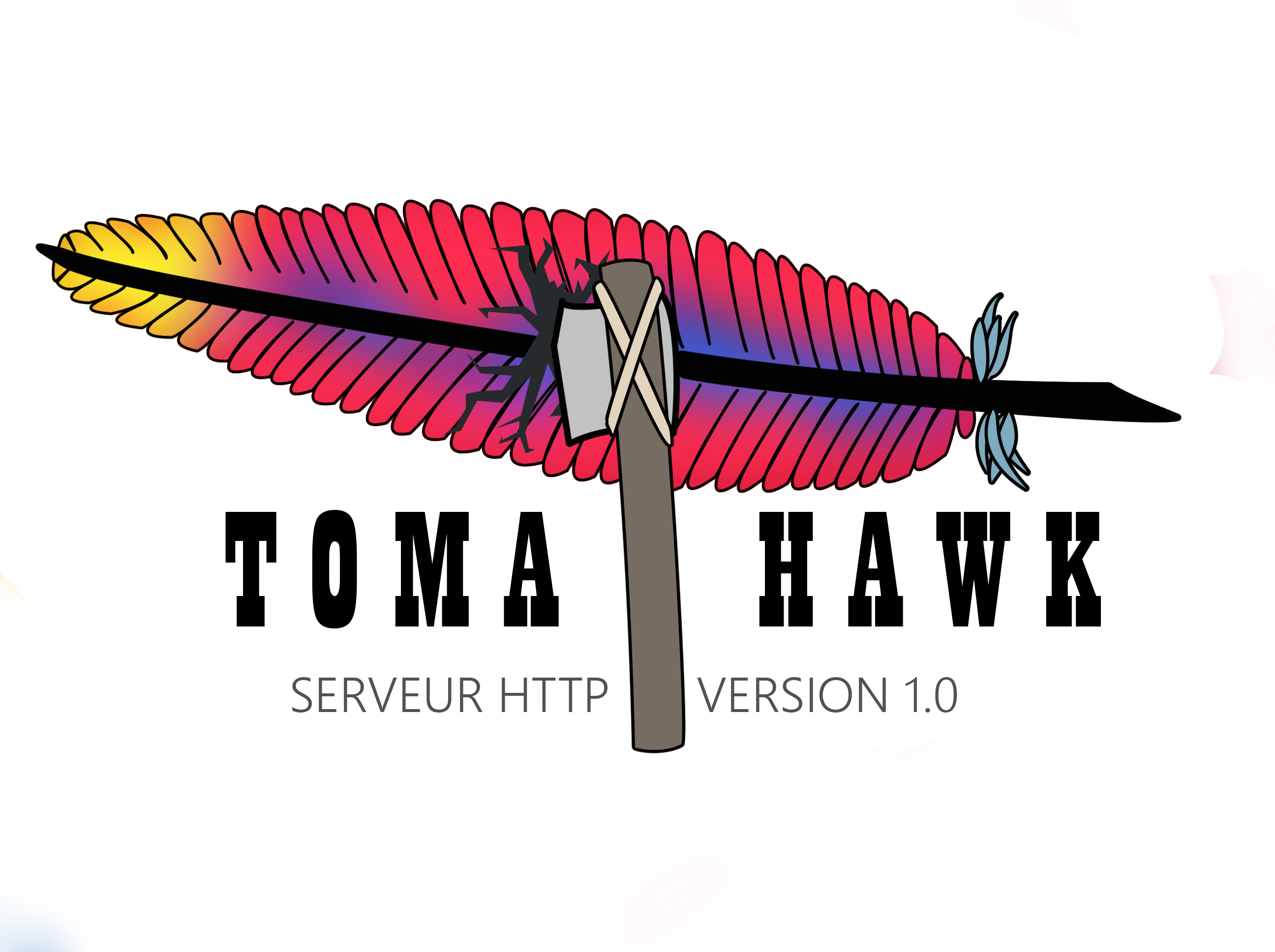
Romain Vidal

Thomas Jacumin

Angelo Lorber

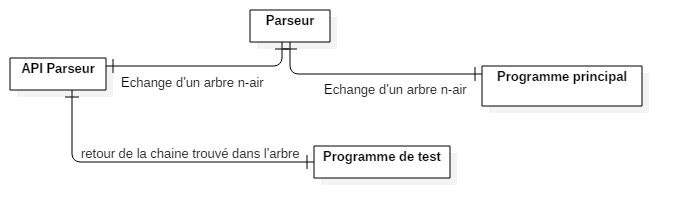
Projet NE 302 – Serveur http

Partie 2 : Parseur http



# Élaboration de l’architecture

## Architecture globale

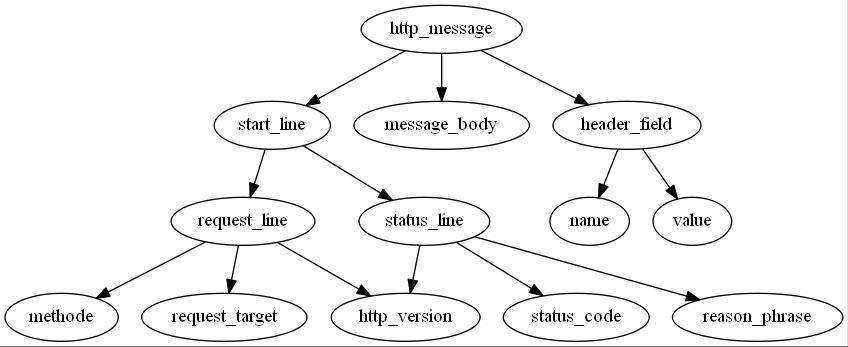


Notre programme principal utilisera un arbre qu’il parcourra pour y étudier les différents champs. Le parseur http étudiera donc une requête passé en paramètre par le programme principal et retournera une structure représentant la requête sous la forme d’un arbre.

L’API échangera de la même manière avec le parseur puis parcourra l’arbre en comparant chaque valeur avec la chaine demandée pour la retourner.

## Le parseur http

Notre parseur va donc prendre en entrée une requête http et la « découper » selon le schéma ci-dessous



Il utilisera des automates pour déterminer l’exactitude de la requête (au niveau syntaxique seulement). Les automates que nous avons utilisés pour cela sont disponibles en annexe I.

# Mise en place du projet

## Mise en place d’un git

Pour la gestion des versions nous avons choisi d’utiliser un git, hébergé sur Git Hub.

Il est disponible à l’adresse suivante :

https://github.com/RomainVidal666/Tomahawk.git

## répartition de la charge de travail

Pour la répartition du travail nous nous sommes retrouvées pour travailler en groupe, notamment pendant la phase de conception, il était indispensable pour nous de pouvoir échanger nos idées.

Pour le codage du parseur nous avons procédé de la même façon, chacun s’occupant d’une fonction. Cela nous a permis d’avancer efficacement et de ne pas rester bloqué sur des problèmes simples.

## Gestion de la documentation

Pour gérer la documentation du code du projet, nous avons choisi d’utiliser Doxygen que nous avions déjà utilisé et qui s’est révélé très efficace.

Il permettra d’avoir une documentation du code correct et de la tenir à jour facilement.

# Implémentation

## Implémentations des automates

L’automate parcourt la chaine à l’aide d’un pointeur sur celle-ci, qu’il fait avancer à chaque lecture. Quand il se trouve sur un état accepteur la sous-chaine est sauvegardée dans l’arbre.

Nous avons écrit une fonction par automates, pour chacune de ces fonctions le comportement est le suivant :

* L’arriver sur un état accepteur termine la fonction.
* Une transition non reconnu termine la fonction.
* La fonction retourne 1 si l’automate est reconnu, 0 sinon.
* Une structure est créée à l’entrée de la fonction et est ajouté à l’arbre.
* Si l’état représente un sous-automate, on appelle la fonction correspondante.

## Implémentation de l’arbre

L’arbre est défini sous la forme d’une liste chainée.

On a défini une structure avec comme attributs :

* Le nom
* un pointeur sur le début de la sous-chaine.
* Un pointeur sur la fin de la sous-chaine.
* Une liste pour stocker un pointeur vers les éléments suivants.

On utilise les fonctions standards de parcourt, d’ajout et de suppression dans un arbre.

## Implémentation de l’api

L’API est assez simple il s’agit de passer la requête au parseur afin récupérer l’arbre puis de le parcourir afin de trouver le champ demandé extraire la sous chaine et enfin, la renvoyer.

# Annexe

## Automate

