



UiT Narvik
Lodve Langesgate 2
8514 Narvik, Norway

Rapport d'élève ingénieur

Stage de 2^e année

Filière 1 : Informatique des systèmes interactifs pour l'embarqué, la robotique et le virtuel.

Conception d'un système de vidéo-surveillance réseau et web autour d'un robot assistant

Présenté par : **Loup RUSAK**

Tuteur ISIMA: M. Romuald AUFRERE

Tuteurs entreprise : M. Hao YU, M. Beibei SHU

Lundi 28 août 2023 11h30

Campus des Cézeaux, 1 rue de la Chégarde, TSA 60125, 63178 Aubière CEDEX, France

Remerciements

Tout d'abord, je tiens tout particulièrement à remercier **l'Université Arctique de Norvège à Narvik (UiT Narvik)** pour m'avoir accepté en tant que stagiaire, au sein de leur département ingénierie industrielle et science de l'informatique, pour mener à bien la réalisation de ce projet de recherche et développement en lien avec leur laboratoire.

Je remercie chaleureusement mon tuteur de stage **M. Hao YU**, professeur agrégé et responsable de programme master du département ingénierie industrielle, pour son accueil et son encadrement. Ayant partagé avec moi la direction novatrice que le département souhaitait prendre pour l'aspect robotique du laboratoire, il m'a fait découvrir le concept d'industrie 4.0 et ses différents aspects.

Dans ce même cadre, je remercie **M. Beibei SHU**, chercheur au département ingénierie industrielle, pour m'avoir encadré tout au long de ce stage, remplaçant M. Yu en raison de son emploi du temps chargé.

Je tiens également à remercier mon tuteur de stage ISIMA **M. Romuald AUFRERE**, pour son accompagnement, son soutien et ses conseils.

Enfin, je remercie mon responsable de filière **M. Mamadou KANTE** ainsi que **Mme Murielle MOUZAT** pour la préparation à la rédaction du rapport de stage.

Table des matières

REMERCIEMENTS.....	2
TABLE DES FIGURES ET ILLUSTRATIONS.....	5
RESUME.....	6
ABSTRACT.....	6
INTRODUCTION	7
I. CONTEXTE DU PROJET.....	8
1.1 PRÉSENTATION DE L'UNIVERSITÉ	8
1.1.1 <i>Son histoire</i>	8
1.1.2 <i>Ses campus et lieux d'études</i>	8
1.1.3 <i>Son organisation</i>	10
1.2 CONTEXTE DE RECHERCHE ET VOLONTES	11
1.2.1 <i>Laboratoire</i>	11
1.2.2 <i>Documentation et formation</i>	11
1.2.3 <i>Définition du sujet</i>	13
II. CONCEPTION ET REALISATION	14
2.1 TECHNOLOGIES CHOISIES	14
2.1.1 <i>Caméra de surveillance</i>	14
2.1.2 <i>Développement web</i>	15
2.1.3 <i>Infrastructure réseau</i>	16
2.1.4 <i>Traitements vidéo</i>	16
2.2 PLANIFICATION DU TRAVAIL.....	17
2.2.1 <i>Planning prévisionnel</i>	17
2.2.2 <i>Planning réel</i>	18
2.3 CONCEPTION.....	19
2.3.1 <i>Maquettage de l'application</i>	19
2.3.2 <i>Infrastructure réseau</i>	23
III. DEVELOPPEMENT ET RESULTATS.....	25
3.1 SITE DE VIDEO-SURVEILLANCE	25
3.1.1 <i>Back-End</i>	25
3.1.2 <i>Front-End</i>	26
3.1.3 <i>Master Detail</i>	28
3.2 SERVEUR-CLIENT ZMQ	30
3.2.1 <i>Version de démonstration</i>	30
3.2.2 <i>Implémentation locale</i>	33
3.2.3 <i>Implémentation distante</i>	36
3.3 RESULTATS ET AMELIORATIONS	37
3.3.1 <i>Résultats</i>	37

3.3.2 <i>Améliorations</i>	40
CONCLUSION	41
LE PETIT POINT ENVIRONNEMENT	42
REFERENCES BIBLIOGRAPHIQUES.....	43
LEXIQUE.....	44

Table des figures et illustrations

Figure 1 : Logo de l'Université Arctique de Norvège et portrait du roi Olav V de Norvège.....	8
Figure 2 : Carte des différents campus de l'Université Arctique de Norvège.....	9
Figure 3 : Organigramme des facultés et domaines d'étude de l'UiT.....	10
Figure 4 : Robot assistant UR10E et camera Pickit 3D	12
Figure 5 : Caméra RS Pro IR CCTV	14
Figure 6 : Logo du Framework Flask	15
Figure 7 : Logo du Framework ZéroMQ.....	16
Figure 8 : Logo de OpenCV	17
Figure 9 : Diagramme de Gantt prévisionnel.....	18
Figure 10 : Diagramme de Gantt réel	19
Figure 11 : Conception page d'accueil sur Figma.....	20
Figure 12 : Conception page détail sur Figma	21
Figure 13 : Conception page vue globale sur Figma	22
Figure 14 : Conception page login sur Figma.....	22
Figure 15 : Schéma infrastructure globale.....	23
Figure 16 : Schéma infrastructure vidéo-surveillance	24
Figure 17 : Exemple de fonction de routage Flask.....	25
Figure 18 : Structure mise en page HTML du site.....	27
Figure 19 : Relation entre Flask Python et code HTML Jinja.....	27
Figure 20 : Structure HTML d'une page de contenu.....	28
Figure 21 : Génération liste dynamique avec Jinja	29
Figure 22 : Schéma représentant l'infrastructure réseau ZMQ de démonstration	31
Figure 23 : Connexion serveurs ZMQ publieur/abonné	31
Figure 24 : Récupération, traitement et envoi vidéo.....	32
Figure 25 : Schéma infrastructure multi-threadée	34
Figure 26 : Exemple de manipulation des threads en Python	34
Figure 27 : Mécanisme de rattrapages des images perdues	35
Figure 28 : Configuration du serveur Waitress	36
Figure 29 : Page de connexion du site	37
Figure 30 : Page d'accueil du site.....	38
Figure 31 : Page de visualisation détaillée du site	38
Figure 32 : Page de visualisation globale du site	39
Figure 33 : Panoramas de la ville de Narvik et des fjords et montagnes alentours.....	42

Résumé

Il m'a été demandé dans un cadre de **recherche technologique**, de réaliser un système de **vidéo-surveillance** pour le **contrôle à distance** d'un **robot assistant**. Ce système comprend une **infrastructure réseau** locale également accessible à distance, ainsi qu'une **interface graphique** permettant la visualisation des caméras.

En étudiant les différentes technologies disponibles et en tenant compte des exigences de l'université, j'ai dû comprendre le travail déjà effectué autour du projet de contrôle à distance, pour ensuite **conceptualiser** et mettre en œuvre les bases d'une nouvelle **infrastructure locale et distante** tout en **maquettant** et développant une **interface web de visualisation**.

Ce travail réalisé en **Python** avec des **Frameworks*** comme **Zéro MQ** ou **Flask**, en **HTML, CSS, Bootstrap** et **JavaScript**, constituera une première avancée dans l'optique d'un projet plus global et sera réutilisé, complété et amélioré pour avoir une infrastructure robotique adaptée à des travaux de recherche étudiante.

Mots-clés: recherche technologique, vidéo-surveillance, contrôle à distance, robot assistant, infrastructure réseau locale et distante, conceptualisation, maquettage, interface web graphique de visualisation, Frameworks*, Python, ZMQ, Flask, HTML, CSS, Bootstrap, JavaScript.

Abstract

In a **technological research** context, I was asked to create a **video surveillance** system for the **cobot*** remote control. This includes a **local and remote network infrastructure** as well as a **graphical user interface** for camera views.

By studying the available technologies and considering the requirements of the university, I had to understand the work already done around this **remote-control** project, to then **conceptualize** and implement the bases of a new infrastructure, while **modeling** and developing a web GUI for visualization.

This work carried out in **Python** with **Frameworks*** such as **ZMQ** or **Flask**, in **HTML, CSS, Bootstrap** and **JavaScript**, will be a first step forward to for a more global project. It will be reused, completed and improved to have an adapted robotic infrastructure to student research.

Keywords: technological research, video-surveillance, cobot* (assistant robot), local and remote network infrastructure, graphical user interface (GUI), remote-control, conceptualization, modelization, Python, Frameworks*, ZMQ, Flask, HTML, CSS, Bootstrap, JavaScript.

Introduction

Dans le cadre de ma deuxième année d'ingénieur à l'ISIMA, j'ai effectué mon stage à l'international au sein de l'Université Arctique de Norvège, et plus précisément dans la ville de Narvik. Pendant quatre mois, d'Avril à Août, j'ai eu la chance de travailler en très grande autonomie sur un projet de recherche informatique dans le département ingénierie industrielle de l'université.

Possédant son propre laboratoire pédagogique, et ayant récemment acquis du matériel robotique tel qu'un robot assistant ou une caméra 3D à usage industriel, le département souhaite développer une infrastructure informatique complète autour de ces machines, dans l'objectif futur de permettre le contrôle et la visualisation à distance. Dans un cas plus général, cette prochaine infrastructure permettra aux enseignants, aux chercheurs ainsi qu'aux élèves de réaliser de multiples travaux de recherche et développement sur le thème de l'industrie 4.0*.

Durant ce stage, c'est dans cette optique que mon travail a consisté à développer une interface web de vidéo-surveillance du robot et les bases de l'infrastructure réseau permettant son fonctionnement. Pour cela j'ai effectué un travail de recherche technologique afin d'imaginer une première ébauche du projet, pour ensuite appliquer ces idées et créer de bout en bout une solution expérimentale adaptée aux besoins de l'université.

La problématique qui m'a suivi durant tout ce stage a donc été de me documenter suffisamment sur les technologies existantes permettant ce type de réalisation, pour conceptualiser de la manière la plus rigoureuse possible, les différents aspects de ce projet ainsi que ses étapes de réalisation.

En parcourant ce rapport, vous explorerez le contexte de ce travail, l'histoire de l'université ainsi que les différentes volontés exprimées sur ce projet d'envergure. Dans un second temps, vous voyagerez au travers des étapes de recherche, de conceptualisation et de réalisation de la solution, notamment par le choix du matériel ou des outils informatiques à utiliser. En troisième lieu, vous découvrirez les coulisses de développement du site web de vidéo-surveillance mais également de son infrastructure réseau, avant de vous dévoiler les résultats obtenus ainsi que les potentielles améliorations à apporter. Pour terminer, je dresserai un bilan de ce projet et de ce stage à l'étranger.

I. Contexte du projet

1.1 Présentation de l'université

1.1.1 Son histoire

Fondée en 1968 et officiellement inaugurée en 1972 par le roi de Norvège Olav V, l'UiT Arctic University of Norway (Université Arctique de Norvège en français, ou Norges Arktiske Universitet en norvégien) est une institution d'état principalement financée par le gouvernement jouissant d'une autonomie considérable tant dans le domaine académique que financier.

L'UiT est une université de recherche de taille moyenne qui contribue au développement fondé sur la connaissance aux niveaux régional, national et international. Elle est devenue la force motrice du nord de la Norvège pour l'avancement de l'éducation et du savoir.



UiT The Arctic
University of Norway



Figure 1 : Logo de l'Université Arctique de Norvège et portrait du roi Olav V de Norvège

L'Université Arctique de Norvège a connu trois fusions au cours de son histoire, ce qui en fait aujourd'hui un pôle international important pour le pays. Le 1^{er} janvier 2009, elle a fusionné avec le Collège universitaire de Tromsø et a été renommée « Université de Tromsø » jusqu'à la fusion avec le Collège universitaire de Finnmark le 1^{er} aout 2013. La dernière fusion a eu lieu le 1^{er} janvier 2016 avec le Collège universitaire de Harstad et celui de Narvik.

1.1.2 Ses campus et lieux d'études

Après ces différentes fusions, l'UiT est devenue une université multi-campus répartie dans tout le Nord de la Norvège. Les principaux campus sont donc situés à Tromsø, Alta, Harstad et Narvik ; campus dans lequel j'ai réalisé ce stage. Elle possède également des départements plus petits dans les villes de Mo i Rana, Hammerfest et Kirkenes.

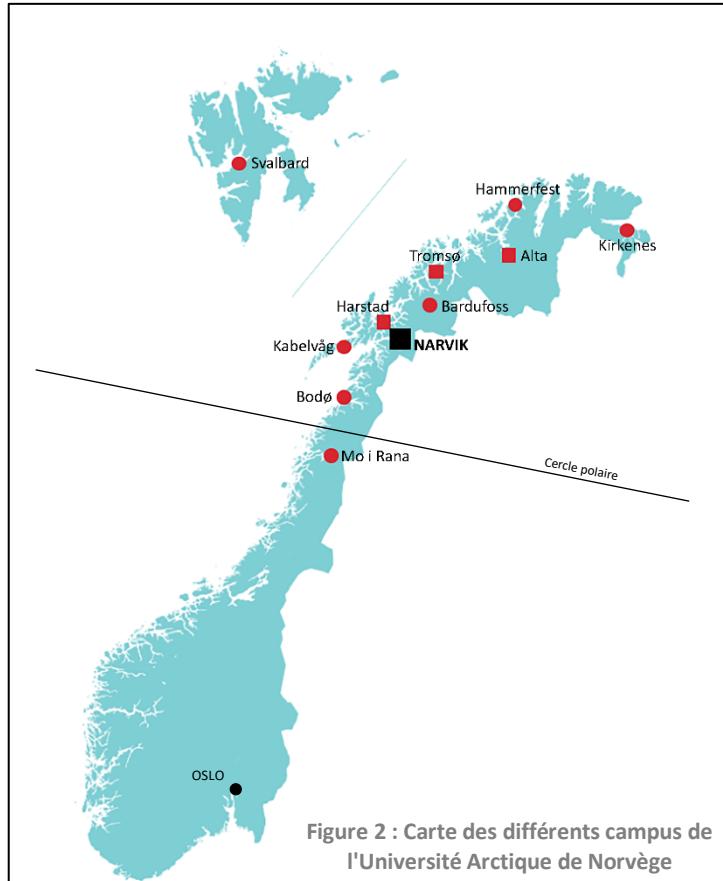


Figure 2 : Carte des différents campus de l'Université Arctique de Norvège

Narvik est la troisième ville la plus importante parmi les villes norvégiennes situées au nord du cercle polaire arctique. En 2019, le nombre d'habitants était d'environ 18 500 et depuis ce nombre grandit petit à petit, attirant de nouveaux résidants chaque année. Son importante activité d'acheminement et d'évacuation par bateau du minerai de fer provenant des mines suédoises, fait de la ville un des ports les plus importants et stratégiques. Pendant le début de la seconde guerre mondiale, les Allemands, grands consommateurs de ce minerai, souhaitaient contrôler cette ligne d'approvisionnement. S'en est suivi une bataille navale et terrestre entre la Norvège, la France, le Royaume Uni et la Pologne d'un côté et l'Allemagne de l'autre. Forte de son histoire, la ville continue de prospérer de cette activité mais tente également de séduire les touristes. Au cœur de la nature, entre montagnes escarpées et grands fjords, cela en fait une destination touristique attractive, pour le ski, la randonnée, le soleil de minuit et les aurores boréales.

L'université Arctique de Norvège à Narvik dispose d'un campus moderne entièrement numérique avec de bonnes installations pour des travaux de laboratoires et des projets. Il accueille chaque année plus de 2 000 étudiants et emploie environ 220 personnes.

1.1.3 Son organisation

L'UiT est la troisième plus grande université de Norvège parmi les huit que le pays compte. C'est également la plus septentrionale du monde. Cette situation géographique, aux portes de l'Arctique, lui impose une certaine mission : protéger l'Arctique et son terrain inestimable. Le changement climatique, l'exploitation des ressources de l'Arctique et les menaces environnementales sont des sujets publics de grande importance, et auxquels l'université s'intéresse tout particulièrement.

L'Université Arctique de Norvège est composée de plusieurs Facultés que vous retrouvez dans la figure ci-dessous. L'expertise de l'université en matière d'enseignement réside principalement dans les domaines scientifiques, mais propose également des matières plus générales comme le sport, l'économie, le droit ou les beaux-arts. Concernant le campus de Narvik, on peut y suivre des enseignements en médecine, en économie, en administration, en sciences de l'aérospatial, en sciences informatiques, ou en sciences électroniques et électriques pour y devenir ingénieur / ingénieur civil avec un parcours international en bachelor, master et doctorat.

Le département où je réalise mon stage est le département ingénierie industrielle de la faculté de technologie et sciences de l'ingénieur.

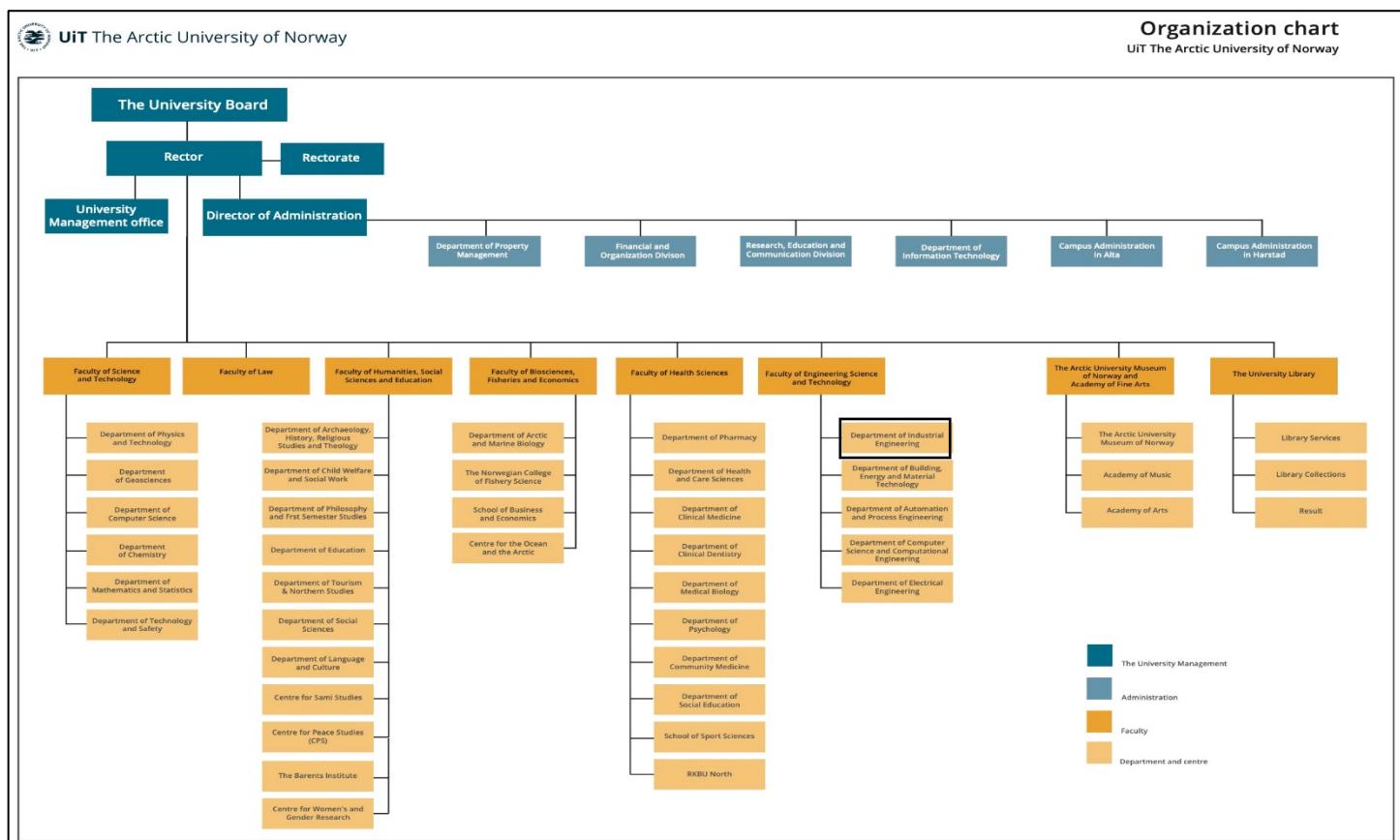


Figure 3 : Organigramme des facultés et domaines d'étude de l'UiT

1.2 Contexte de recherche et volontés

1.2.1 Laboratoire

Le département Ingénierie industrielle de l'université de Narvik possède, pour l'enseignement et la recherche, son propre laboratoire au sein des bâtiments de l'UiT.

On y retrouve plusieurs espaces de travail distincts avec des machines de fabrications comme des découpeuses, fraiseuses ou imprimantes 3D. On peut aussi dire que ce laboratoire est une sorte de fablab* car il comprend également des espaces de coworking permettant d'animer des cours ou donnant la possibilité aux étudiants de mener des projets de recherche divers et variés.

Dans ce grand espace, le département a récemment fait l'acquisition d'un « cobot* » ; un robot assistant UR10E de la marque Universal Robots et a aménagé un espace de travail évolutif et collaboratif autour de celui-ci. L'objectif à long terme étant de créer un système de logistique et de fabrication intelligent selon le modèle de l'industrie 4.0, le département veut construire et développer une infrastructure technologique complète autour de ce robot. Cela permettra aux étudiants d'effectuer des tests et des recherches sur les outils et techniques de cette nouvelle industrie et ainsi participer au développement continu de ce système.

L'objectif à court terme auquel j'ai participé durant mon stage est donc de réaliser les fondements de cette infrastructure et construire dans le même temps un système de sélection automatique ainsi que de contrôle à distance du robot.

Lors de mon arrivée et durant la présentation que l'on m'a faite, on m'a spécifié que certaines des composantes voulues pour ce système ont déjà ou sont en cours d'être développées par des étudiants en master dans le cadre de leur projet annuel. A cette occasion, j'ai rencontré Caleb, étudiant master, qui a travaillé sur un système de communication à distance de valeurs de mouvement pour le robot avec interface graphique et via un serveur utilisant le standard de communication industriel OPCUA.

1.2.2 Documentation et formation

Après avoir rencontré mon tuteur et mes responsables de stage et assisté à des réunions où ils m'ont expliqué le contexte global du projet que je vous ai détaillé ci-dessus, j'ai commencé à me documenter et à me former sur le matériel et les technologies que je serai amené à utiliser.

La première étape a donc consisté à étudier le cœur de ce futur système : le robot assistant UR10E. Pour comprendre au mieux la volonté du département d'ingénierie industrielle j'ai eu à comprendre son fonctionnement, connaître les possibilités qu'il offre et bien sûr savoir comment le manipuler et l'utiliser. J'ai donc suivi une formation interactive en ligne du robot, directement sur le site Universal Robots.

Ce cours, divisé en 8 rapides modules et disposés dans un ordre logique, m'ont permis de :

- Connaitre la construction matérielle du robot en lui-même, ses différents moteurs et axes de mouvements, son bloc de connexion et sa tablette de contrôle.
- Savoir comment connecter des entrées et sorties directement au robot tels que des capteurs ou des convoyeurs.
- Comprendre comment installer et configurer différents outils sur la tête du robot.
- Créer des programmes simples pour effectuer des mouvements et interagir et contrôler des appareils externes.
- Apprendre et mettre en place les réglages de sécurité nécessaires pour différents types d'opérations.
- Optimiser les déplacements et actions du robot pour gagner en temps et en efficacité.

J'ai logiquement dû par la suite me documenter rapidement sur OPCUA. Open Connectivity Unified Architecture est un standard d'échange de données flexible, fiable et sécurisé, adapté aux besoins d'interopérabilité des systèmes industriels. Il m'a donc été donné d'étudier rapidement une bibliothèque Python sur GitHub se servant de ce standard afin de comprendre dans les grandes lignes son fonctionnement.

La deuxième étape a été de m'intéresser au système de cueillette automatique de pièces par le robot. Pour cela, l'université a également acheté récemment une caméra spécialisée dans le domaine : la Pickit L-SD de la marque Pickit 3D. Afin de comprendre le fonctionnement de celle-ci et comment l'intégrer à leur système, certains professeurs, doctorants et masters du département d'ingénierie industrielle ont organisé une formation vidéo avec l'entreprise à laquelle j'ai été convié. Durant cette journée de formation, j'ai pu comprendre les différentes applications de cette caméra ainsi que beaucoup de théorie sur son fonctionnement dans plusieurs situations typiques. J'ai ensuite suivi des tutoriels supervisés de mise en place et installation de la caméra avant de prendre en main l'interface web de programmation de la caméra.



Figure 4 : Robot assistant UR10E et camera Pickit 3D

1.2.3 Définition du sujet

Durant cette formation de début de stage, je peux dire que j'ai vu dans la globalité tous les aspects du système à développer, il me restait à avoir un contexte de travail cadré afin de développer une partie de ce système.

Mais avant cela, étant éventuellement amené à travailler en autonomie dans le laboratoire, il m'a été nécessaire d'effectuer une courte formation de sécurité dans un environnement industriel laboratoire. Les moyens mis à notre disposition pour nous protéger lors d'un travail manuel, les règles de sécurité fondamentales vis à vis des diverses machines présentes, le panneau de contrôle électrique général, la gestion de l'alarme ou encore du monte-charge et de son treuil ; toutes ces informations m'ont été communiquées au plus vite.

Concernant le sujet précis de mon stage, il m'a été communiqué par mon tuteur, deux sujets différents étant en réalité deux axes d'améliorations et de volontés pour le système, conceptualisés en tenant compte du travail qui a déjà été réalisé en amont de ma venue.

Le premier sujet que l'on m'a proposé consiste à effectuer la connexion et l'installation initiale de la caméra Pickit 3D avec le robot UR10E et ensuite avec le serveur local OPCUA. Tout cela dans le but de rendre inter-opérant tous les robots du laboratoire, notamment pour le futur système de gestion à distance.

Mais fort de mon parcours et de mes expériences passées dans le développement web et logiciel, ma préférence est allée vers ce deuxième sujet ; le développement d'une interface web de visualisation de caméras de surveillance autour du robot. Couvrant différents angles de vues, ces caméras doivent permettre de s'assurer de la sécurité des humains et du robot au sein du laboratoire. Bien entendu, ce sujet faisant également partie du futur système de gestion à distance, les flux vidéo en provenance de ces caméras doivent être envoyées vers le Cloud pour que l'utilisateur puisse les visualiser. Cette installation nécessitant une infrastructure locale et distante pour satisfaire les exigences, on m'a également demandé dans un cadre de recherche de conceptualiser et de poser les bases de celle-ci, non seulement pour la vidéo surveillance mais surtout dans sa globalité et dans le but d'accueillir d'autres composantes.

J'ai donc dû durant ce stage, rechercher ce qui a déjà été fait sur ce thème, m'en inspirer pour conceptualiser une solution, développer ses fondements et enfin concilier et joindre différentes technologies aux fonctions variées dans une optique d'évolutivité.

II. Conception et réalisation

2.1 Technologies choisies

2.1.1 Caméra de surveillance

L'élément primordial et constitutif d'un système de vidéosurveillance réside dans les caméras de surveillance. Il est important suivant les besoins de savoir quel type de caméra on souhaite car il en existe différents types : les caméras espionnes pour une surveillance discrète, les caméras Wifi qui privilégient un accès à distance via un smartphone ou une tablette, et enfin les caméras filaires qui sont destinées à plus de cas d'utilisation comme une intégration dans un système plus vaste. Il faut aussi réfléchir à la fonction première de ces caméras et cela va également dicter le mode d'alimentation et d'enregistrement.

Dans mon cas, l'université a fait l'acquisition de 3 caméras filaires réseaux que j'ai pu utiliser pour mon travail. Ces caméras sont des IR CCTV de la marque RS Pro. Elles sont destinées par leur conception à être intégrées dans un système contrôlé à distance et diffusé en circuit fermé. Elles peuvent fonctionner autant en intérieur qu'en extérieur et également de jour comme de nuit grâce à des capteurs infrarouges. Le capteur photo de la caméra est de 2 millions de pixels ce qui lui permet de restituer une image d'une qualité full HD soit 1920 par 1080 pixels.

A l'intérieur de cette caméra on retrouve ni plus ni moins qu'un système Linux embarqué gérant les différents composants de la caméra mais également les interfaces de communication externes. Elles sont capables de diffuser simultanément et individuellement 3 flux vidéo. Ce type de caméras est aussi directement livré avec la dernière version du protocole standard ONVIF, permettant de communiquer sans soucis avec d'autres appareils de surveillance de constructeurs différents. Tout cela en fait des caméras simples et efficaces pour un système domestique comme commercial.



Figure 5 : Caméra RS Pro IR CCTV

Au sein du laboratoire de l'université, ces caméras devront être placées de manière à couvrir les angles de vue les plus judicieux pour la surveillance du robot mais également la sécurité des utilisateurs. Une première caméra servira à visualiser la zone du robot par la droite de celui-ci. Logiquement, une deuxième sera disposée du côté opposé et donc à gauche. Enfin, la troisième camera sera quant à elle focalisée sur les mouvements du robot depuis le dessus. Bien évidemment il existe d'autres angles de vues intéressants à couvrir comme une vue de devant ou encore une vue plus large de toute la zone de manipulation et ses occupants. La conception de ce système de vidéo-surveillance doit être évolutive pour pouvoir intégrer de nouvelles caméras dans le futur.

2.1.2 Développement web

Pour réaliser le développement du site web et de l'infrastructure en général permettant la surveillance, on m'a demandé d'utiliser principalement le langage de programmation Python. Certaines des autres composantes de l'infrastructure du laboratoire ayant déjà été réalisées par d'autres personnes dans ce langage, cela permettra d'avoir une certaine harmonie et une intégration simplifiée.

Concernant le développement du site web en lui-même, les langages primaires utilisés sont HTML et CSS, mais ceux-ci restent limités dans leur état pur. C'est pour cela qu'une des exigences pendant ce stage a été d'utiliser le Framework* Flask en Python pour développer le cœur du site. Flask est un Framework* open-source de développement web en Python, très léger et donc performant. Il a pour objectif de garder un noyau d'application simple mais extensible à souhait. En effet il n'intègre pas par défaut de système d'authentification, pas de base de données, ni d'outils de validation de formulaires. Cependant, de nombreuses extensions Python permettent d'ajouter facilement ces fonctionnalités au besoin.



Figure 6 : Logo du Framework Flask

Nativement, Flask intègre et dépend de deux autres librairies de code : Jinja et WSGI Werkzeug. Jinja est un moteur de template en langage Python permettant donc de générer des patrons réutilisables et personnalisables de n'importe quel type de fichier pouvant être balisé, comme des pages HTML. Contrairement à ces concurrents, il permet plus de personnalisation via des filtres, des tests, des expressions et permet au développeur d'appeler des fonctions avec des arguments sur des objets. WSGI Werkzeug est une sorte de boîte à outils permettant à Flask de communiquer et d'échanger avec les interfaces réseaux du système informatique de déploiement via ce qu'on appelle le routage, afin de réagir aux requêtes et réponses web.

Flask est en concurrence directe avec un autre Framework* appelé Django. Tous deux ont des avantages et des inconvénients, si bien que les professionnels ont du mal à déterminer lequel utiliser en

fonction de leurs besoins. Pour un projet assez petit comme celui que j'ai eu à développer, Flask a semblé être plus simple et plus adapté à une application web de vidéo-surveillance.

2.1.3 Infrastructure réseau

La partie le plus importante du développement de ce système de vidéo-surveillance est l'infrastructure réseau qui permettra de faire le lien entre le matériel de vidéo en local, le site web distant de visualisation ainsi que l'utilisateur final de celui-ci. Depuis le début du projet, je savais qu'une communication multidirectionnelle allait s'effectuer entre deux entités distinctes, d'un côté une transmission vidéo et de l'autre des requêtes et des réponses. D'où la nécessité de choisir une technologie de communication performante et adéquate. Pour cela on m'a orienté vers un Framework* appelé ZeroMQ (ou ZMQ).

ZMQ est une bibliothèque réseau de messagerie asynchrone à haute performance disponible dans une multitude de langages de programmation. Elle peut être utilisée pour des applications distribuées, c'est-à-dire avec de multiples processus s'exécutant sur des machines distantes, comme pour des applications concurrentes avec différentes entités pouvant communiquer ou non entre elles. ZMQ fournit des files d'attentes de messages ainsi que des sockets, qui s'occupent du transport de l'information sur le réseau via divers protocoles de communication. Ses différents modèles et architectures d'entrées/sorties reposant sur les sockets asynchrones, permettent de créer des applications parallélisées évolutives et performantes. Outre la performance et la versatilité qu'offre ZMQ, le fait que Microsoft, Samsung ou encore Facebook garantie une grande communauté d'utilisateurs et une grande variété de cas d'utilisation.



Figure 7 : Logo du Framework ZéroMQ

Utiliser ce Framework* dans mon cas afin de constituer le cœur de l'infrastructure a semblé particulièrement judicieux car la récupération et le traitement des flux vidéo doit se faire **en permanence**. En revanche le traitement sur le site web, l'affichage et son actualisation doivent s'effectuer seulement lorsque l'on a une demande de la part d'un utilisateur.

2.1.4 Traitement vidéo

Enfin, comme les flux vidéo envoyés par les caméras ont besoin d'être traités pour ce projet, que cela soit pour récupération, envoi, réception ou affichage, il a fallu trouver les outils nécessaires disponibles en Python. Un des doctorants présents à l'université, travaillant sur de la vision assistée par ordinateur, m'a orienté vers la librairie OpenCV, que j'ai déjà manipulé à l'ISIMA dans des travaux pratiques du même domaine.

OpenCV est une bibliothèque libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel. Elle met à disposition de nombreuses fonctionnalités très diversifiées, allant d'opérations classiques en traitement bas niveau des images à des traitements algorithmiques plus lourds pour le traitement vidéo ou encore l'apprentissage. OpenCV étant une des bibliothèques les plus connues et les plus polyvalentes dans le domaine, c'est un choix judicieux et sûr pour ce projet.



Figure 8 : Logo de OpenCV

2.2 Planification du travail

Durant ce stage, j'ai été en totale autonomie, que ce soit dans ma réflexion de la conception et mes choix de développement. Étant donné qu'aucune tâche précise ni d'objectif à court terme ne m'étaient fixés, et donc le processus de développement plutôt linéaire, je n'ai pas travaillé en méthode agile mais j'ai préféré appliquer la méthode cascade tout en me fixant des objectifs à atteindre à la fin de chaque semaine.

2.2.1 Planning prévisionnel

Après m'être approprié le sujet notamment en effectuant de multiples recherches, j'ai discerné plusieurs étapes ou phases successives pour la réalisation de ce projet. Débutant ce projet de zéro, j'ai d'abord estimé nécessaire de commencer par une importante phase de recherche et de conception. Dans la suite logique, une fois les idées établies, débuter le développement du cœur du site web de vidéosurveillance et de ses principales fonctions statiques. Puis développer les bases de l'infrastructure réseau pour récupérer les flux vidéo et les afficher dynamiquement pour enfin déployer cette infrastructure en local et dans le cloud, et peaufiner le tout pour améliorer la stabilité et réduire le nombre de bugs.

Vous pouvez voir sur le diagramme prévisionnel ci-après les différentes étapes expliquées précédemment avec le détail de certaines opérations intermédiaires :

GANTT PRÉVISIONNEL

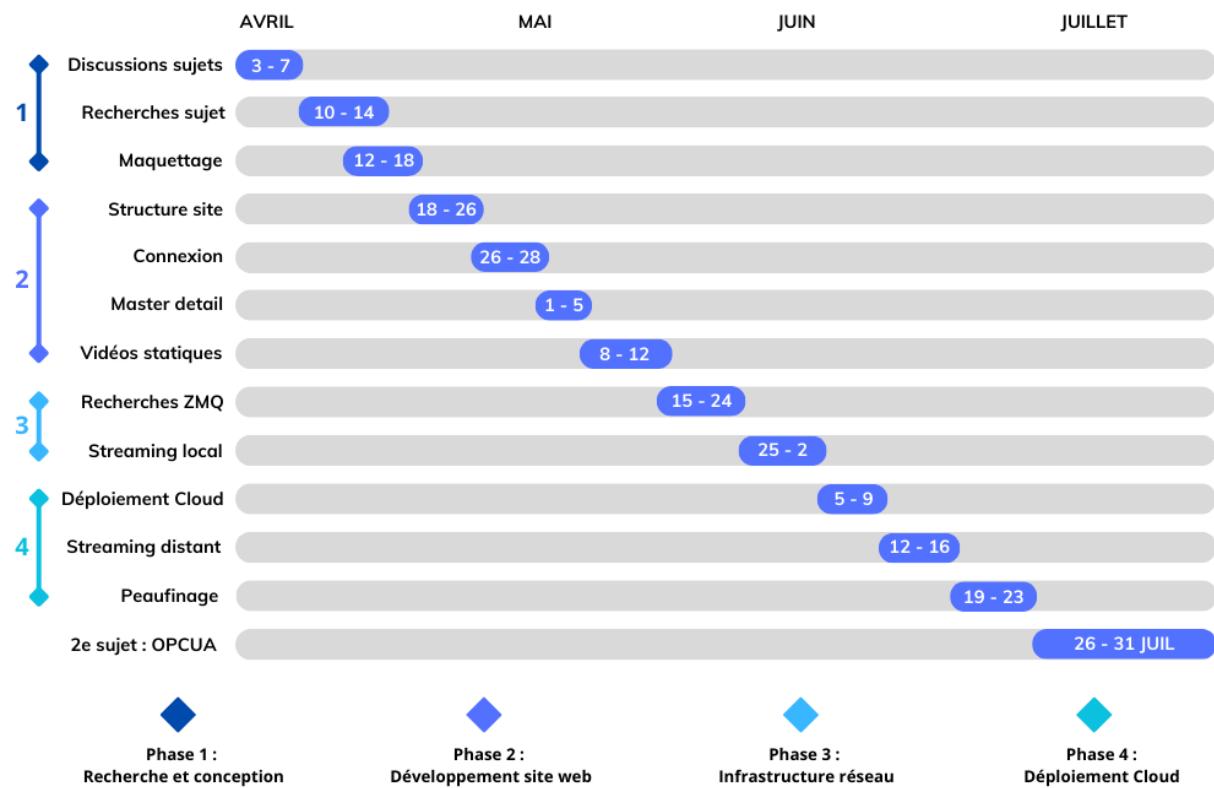


Figure 9 : Diagramme de Gantt prévisionnel

2.2.2 Planning réel

Comme vous pourrez le remarquer sur le diagramme réel ci-dessous, certaines des étapes du projet ont été considérablement rallongées par rapport à mes prédictions. Il y a eu principalement deux situations dans lesquelles cela a été le cas :

- Le début de mon stage, étant malheureusement tombé la même semaine que les fêtes de Pâques norvégiennes. Mes encadrants ainsi que la plupart du personnel de l'université étant en vacances, certains points du sujet n'étaient pas complètement établis, comme les technologies à utiliser. De plus, du fait des petites formations que j'ai réalisées par la suite, la première phase de recherches et de conception a donc été retardée de quelques jours.
- Pendant le développement, étant en totale autonomie, j'ai quelques fois dû revoir certains aspects de conception et de réalisation. Comme je vais l'expliquer par la suite, j'ai à plusieurs reprises buté sur des problèmes ou bien le résultat intermédiaire n'était pas satisfaisant. Tout cela a contribué à rallonger les délais de façon générale, même si certaines tâches ont demandé moins de temps que prévu.

GANTT RÉEL

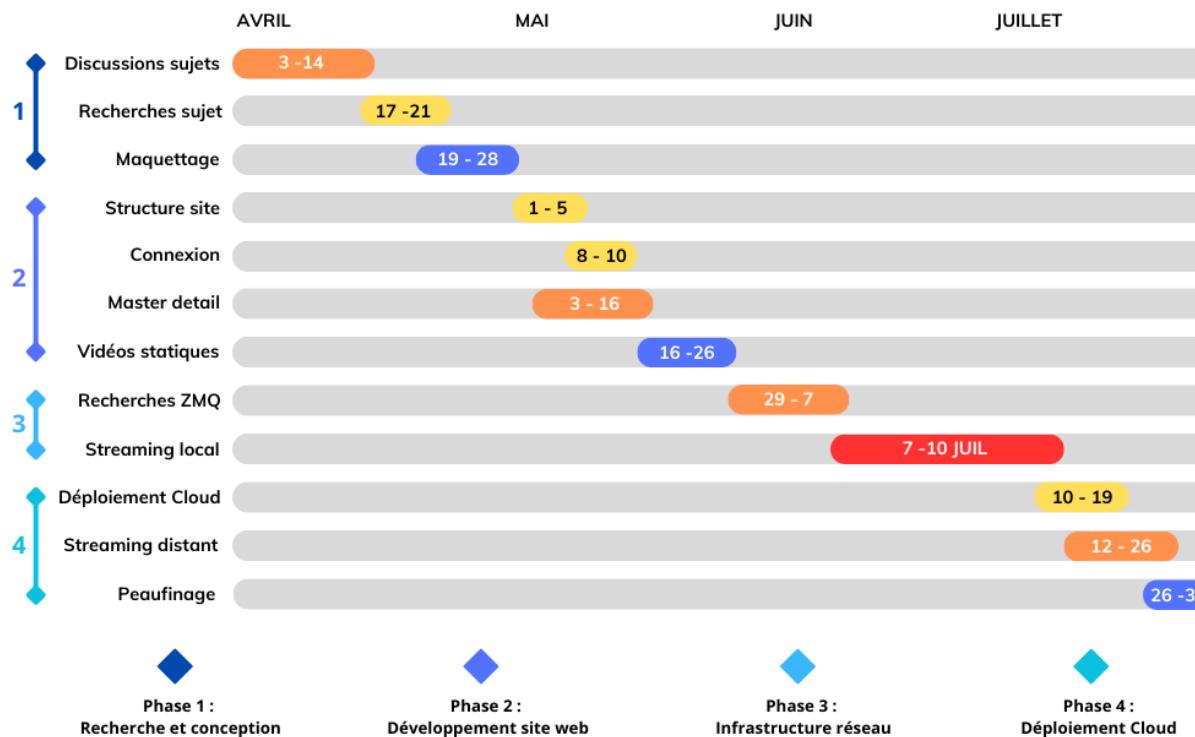


Figure 10 : Diagramme de Gantt réel

2.3 Conception

2.3.1 Maquettage de l'application

Une fois le sujet de stage précisé, les formations effectuées et, en parallèle des recherches sur Flask, j'ai réalisé un premier maquettage de l'application. Pour cela, j'ai cherché sur internet des exemples d'agencement d'éléments d'interfaces graphiques afin de constituer une expérience utilisateur adaptée.

Il a très vite été nécessaire de réaliser une liste sous forme de cahier des charges regroupant les idées de conception ainsi que les fonctionnalités jugées indispensables et celles secondaires. Il était important d'avoir :

- Une page d'accueil. Possédant pour le moment trois caméras avec des angles de vue différents, et sûrement plus dans le futur, il est nécessaire d'avoir un affichage simple et compréhensible permettant de les répertorier et de connaître facilement leur fonction ainsi que leur état de fonctionnement.
- Un affichage permettant la visualisation en premier plan d'une caméra, dans le but d'observer précisément un angle de l'espace du robot et ce qu'il s'y passe. Pour assurer une certaine

polyvalence, on doit pouvoir changer de point de vue rapidement sans forcément recharger une page entière du site.

- Un affichage secondaire global donnant la possibilité d'observer rapidement ce qu'il se passe dans la scène, sous plusieurs angles en même temps et tous si possible. Cette interface doit permettre d'observer les mouvements du robot dans son espace et la présence des usagers dans la globalité de l'espace couvert par les caméras.

J'ai par la suite matérialisé ces premières idées et cherché une structure globale simple d'utilisation et évolutive. Le logiciel de design Figma m'a permis de réaliser ces essais facilement et proprement, certains modèles d'interfaces graphiques étant déjà présents au cœur de celui-ci.

2.3.1.1 Page d'accueil

Afin de fluidifier au maximum la navigation entre les différents affichages, créer une barre d'onglets commune semblait la solution la plus ergonomique. Celle-ci contiendrait donc le nom du site associé au logo de l'université et enfin les 3 onglets : accueil, vue centralisée et vue multiple.

Pour remplir la page d'accueil, j'ai essayé de reprendre le visuel du robot UR du laboratoire comme base pour y superposer les points de vue de caméra disponibles. Comme vous pourrez le voir sur l'image ci-dessous, nous avons une vue schématisée du dessus du robot et de son espace de manipulation sur laquelle sont apposées des logos de caméras avec en leur centre un code couleur pour définir leur état de fonctionnement. La schématisation du cône de champ de vision sur chaque caméra permet à l'utilisateur de comprendre simplement sa position dans l'espace. De plus, ces éléments doivent être cliquables et emmener directement à la vue en détail de la caméra en question.

Le style général de l'application est simple et les couleurs choisies sobres, pour permettre une navigation intuitive et adaptée au plus grand nombre de personnes, sans être austère.

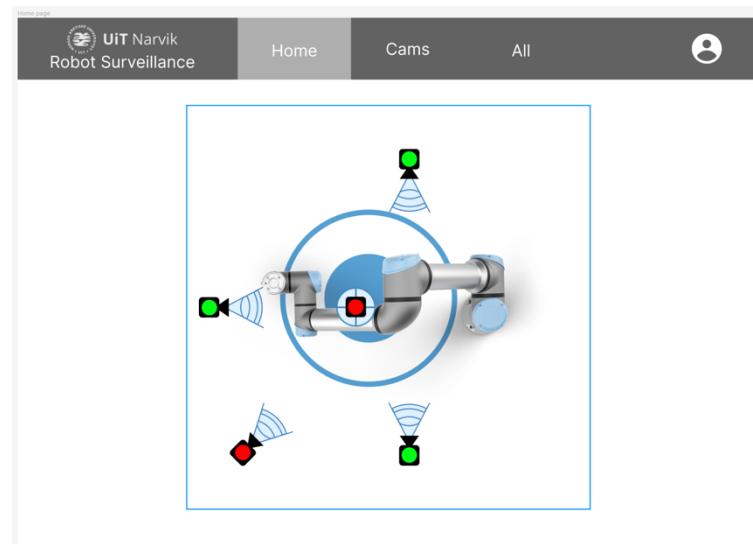


Figure 11 : Conception page d'accueil sur Figma

2.3.1.2 Page de vue en détail

Afin de visualiser précisément le flux vidéo d'une des caméras et de changer rapidement et simplement sans naviguer dans les pages du site, j'ai imaginé un onglet avec un affichage sous forme de master-detail. Dans la conception d'une interface utilisateur, un master-detail affiche une liste maître et les détails de l'élément de la liste actuellement sélectionné. La liste, disposée en colonne sur la partie gauche de l'écran, affichera les caméras disponibles par leur nom, faisant référence à l'angle de vue de celles-ci. Comme vous pourrez le voir sur l'image ci-dessous, en cliquant sur une des caméras de la liste, le flux de la caméra s'affiche avec autour les mêmes informations pertinentes qu'auparavant, comme le nom de la caméra, le statut de celle-ci ainsi que la date et l'heure. En complément de cette vue j'ai jugé bon d'ajouter la fonctionnalité secondaire de prendre une capture d'écran ou vidéo du flux affiché et de retrouver les fichiers dans l'explorateur du système d'exploitation. Il est ensuite possible de cliquer sur une autre caméra de la liste et tout le détail et les informations se mettent à jour sans délais.

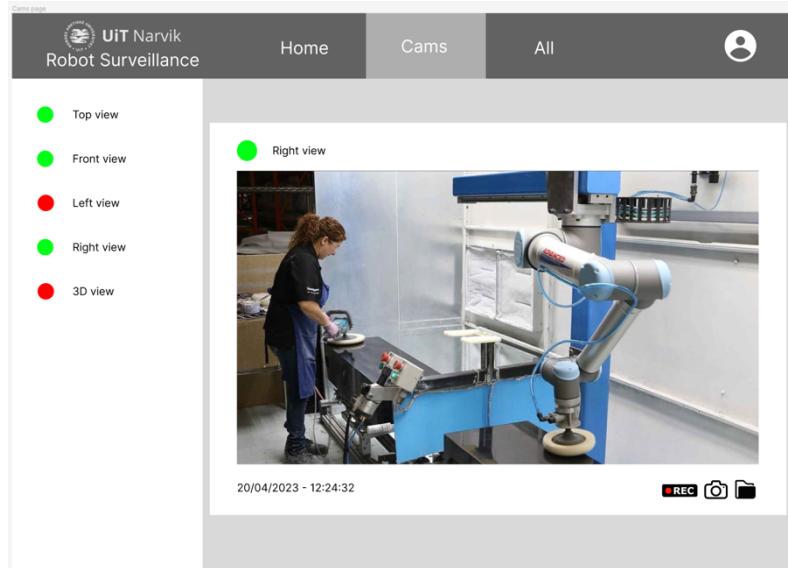


Figure 12 : Conception page détail sur Figma

2.3.1.3 Page de vue globale

Concernant le dernier onglet, j'ai voulu donner la possibilité de voir la scène sous le plus d'angles possibles en même temps. J'ai donc matérialisé une grille défilante dans un sens vertical dans laquelle jusqu'à deux flux s'affichent sur une même ligne et sous la même forme que dans la page de détail. En défilant vers le bas l'utilisateur verra naturellement d'autres lignes affichant également d'autres flux vidéo de caméras, donnant ainsi une vision globale et simultanée mais sans détail en raison de la petite taille des cadres vidéo.

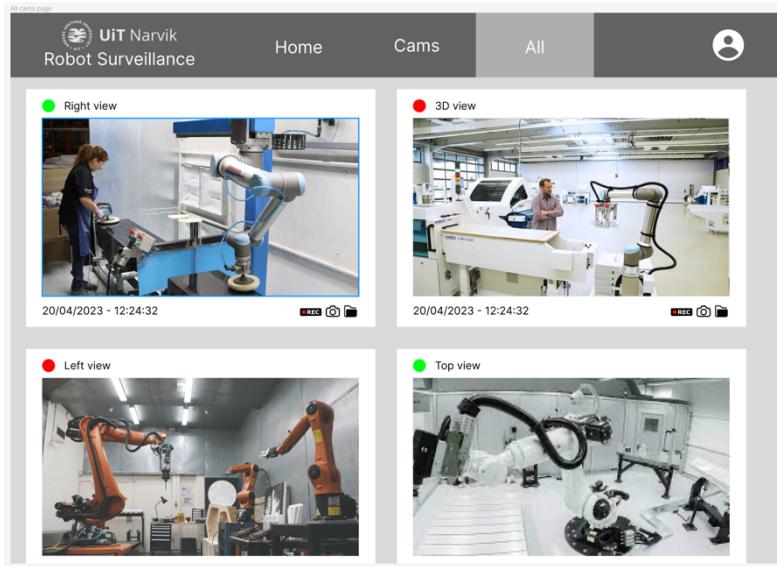


Figure 13 : Conception page vue globale sur Figma

2.3.1.4 Page de login

Pour des soucis de sécurité, j'ai également proposé une page d'authentification avant d'accéder au contenu du site web. Étant donné que les flux vidéo sont internes au laboratoire et que l'on peut potentiellement observer des éléments privés ou du personnel de l'université, j'ai jugé important de sécuriser l'accès au site aux seuls identifiants autorisés.

Comme vous pouvez le voir sur l'image ci-dessous, un formulaire d'authentification via identifiant et mot de passe s'affiche au centre de l'écran, entouré par un fond photo de l'université de Narvik au cœur des paysages de Norvège. Et tant que vous n'êtes pas connecté, aucun onglet ne s'affiche et aucune navigation dans le site n'est possible.

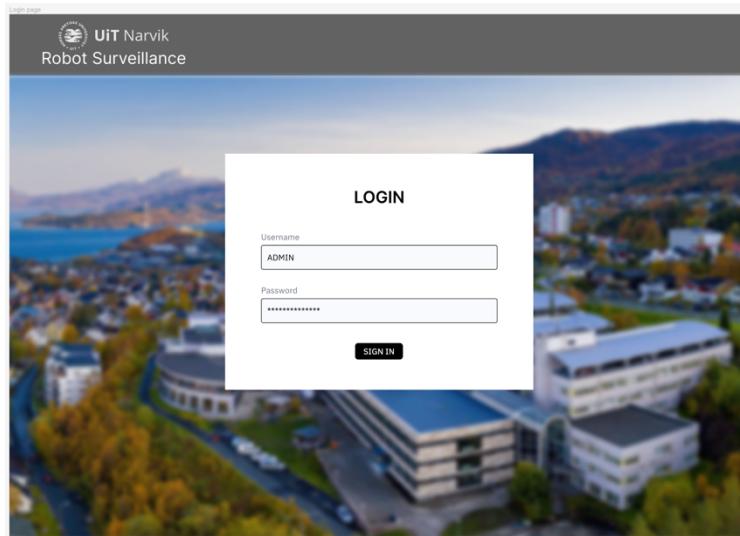


Figure 14 : Conception page login sur Figma

2.3.2 Infrastructure réseau

Pendant cette phase de conception, il a aussi fallu conceptualiser le fonctionnement interne de l'application et en priorité le fonctionnement de l'infrastructure serveur-client entre cloud et laboratoire. Lors de ma période de formation et de définition du sujet, on m'a partagé un schéma expérimental de l'architecture matérielle et logicielle voulue pour la prochaine infrastructure technologique du laboratoire.

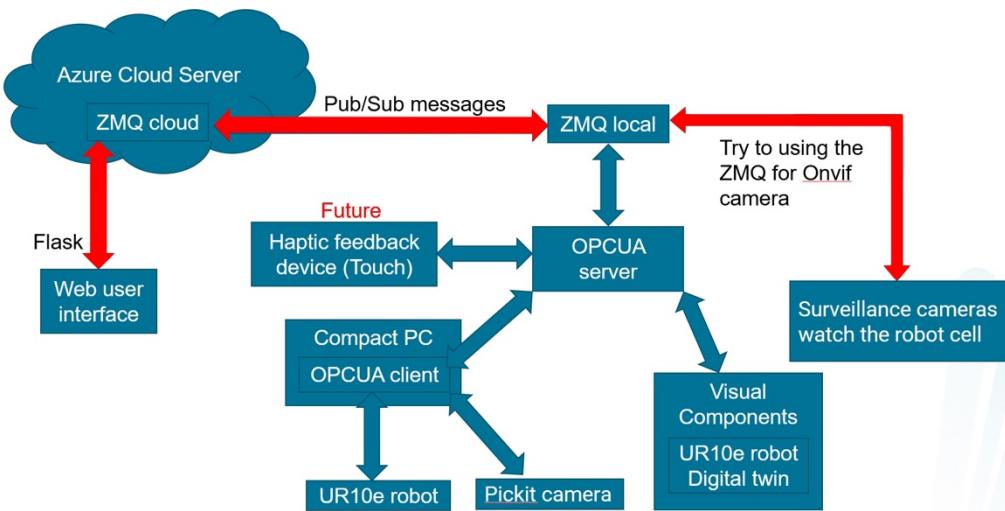


Figure 15 : Schéma infrastructure globale

Comme vous pouvez le constater, on retrouve bien sur le schéma ci-dessus, les volontés exprimées dans la présentation du contexte et de mon sujet, à savoir la mise en place d'un serveur OPCUA permettant de communiquer avec le robot et la caméra Pickit, ainsi qu'une infrastructure réseau connectant le laboratoire et internet pour le contrôle à distance des machines et la vidéo-surveillance. Le travail que j'ai eu à réaliser durant ce stage est ici matérialisé par les flèches rouges.

Pour mettre en œuvre cette infrastructure, il a été imaginé de réaliser deux serveurs ZMQ :

- Un premier serveur local au laboratoire, assurant le lien entre les caméras, le serveur OPCUA local et le Cloud. Il doit tout d'abord récupérer les flux vidéo des caméras, les traiter et les envoyer au Cloud. Dans un second temps ce sera également lui qui fera le lien avec le serveur OPCUA local en transmettant des données plus complexes pour le robot, la caméra et les futurs ajouts comme l'interface de contrôle tactile.
- Un deuxième serveur ZMQ, cette fois dans le Cloud, chargé de récupérer les données vidéos envoyées depuis le serveur local et de faire le lien avec l'appli web Flask pour afficher les flux de vidéo-surveillance. A l'avenir ce serveur devra également lier tout autre ajout concernant le contrôle à distance.

J'ai donc repris l'idée initiale et, d'après mes recherches, j'ai pu modéliser un schéma plus détaillé de cette partie et réaliser une version de démonstration locale sur mon ordinateur.

Les deux serveurs doivent communiquer sous le principe de publieur/abonné. Premièrement, le serveur ZMQ local doit exécuter en continu un fichier Python chargé de récupérer avec OpenCV les différents flux vidéo depuis les caméras, faire les opérations de conversion ou de compression nécessaires pour l'envoi et enfin les publier sur le réseau à quiconque s'abonne à cette diffusion. Il a fallu ensuite choisir un protocole de transport de données entre TCP*, UDP* et d'autres. J'ai choisi d'utiliser TCP* car c'est un protocole plus fiable et sécurisé et il fonctionne en mode connecté, ce qui implique une connexion et la création d'une session entre les deux parties, ce qui réduit le risque d'interception des données par un tiers. Dans un second temps, le serveur ZMQ distant établit la connexion au flux de diffusion et récupère en continu les données vidéos. Ensuite, pour que le site web Flask puisse afficher les vidéos, le serveur ZMQ distant va servir de relais et communiquer les données directement à l'application. Le site de vidéo-surveillance quant à lui se chargera d'effectuer les transformations inverses sur les données vidéo et les affichera à la demande de l'utilisateur. J'ai résumé toute cette réflexion et ses éléments techniques dans le schéma ci-dessous :

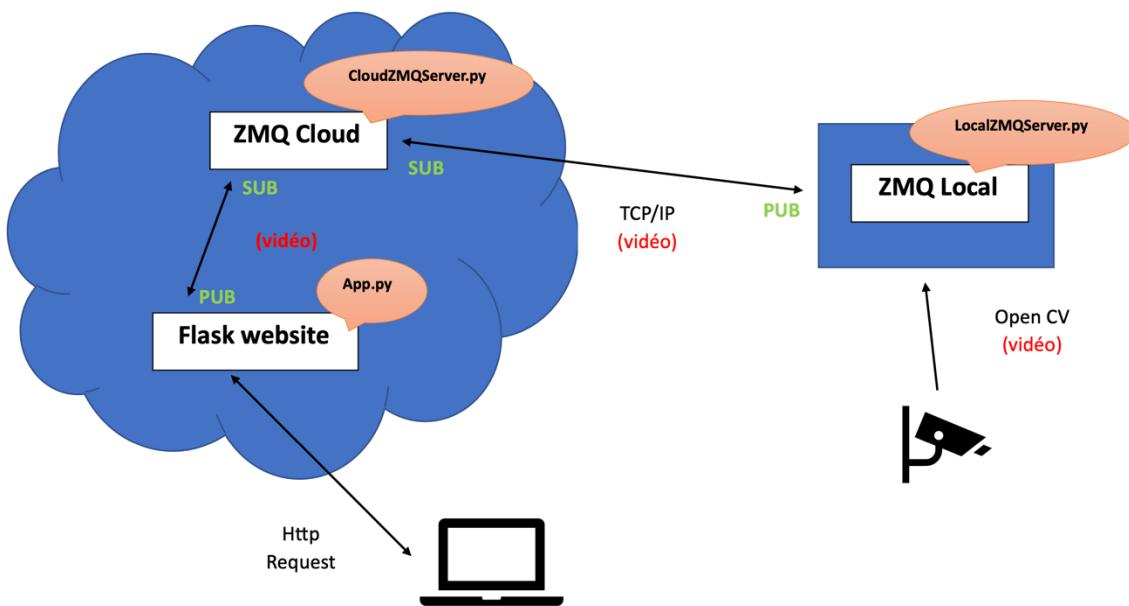


Figure 16 : Schéma infrastructure vidéo-surveillance

Cette conception a bien entendu été amenée à évoluer et quelques fois à être revue lors du développement. J'explique les raisons et les problèmes rencontrés dans la partie suivante.

III. Développement et résultats

3.1 Site de vidéo-surveillance

Une fois cette première phase de recherche et conception bien avancée, j'ai débuté le développement du site internet de vidéo-surveillance aussi bien en réalisant le front-end que le back-end de l'application.

3.1.1 Back-End

Le back-end d'un site est tout de ce qui a la charge de l'aspect technique et fonctionnel, les fonctionnalités de celui-ci en faisant partie. Au premier abord, il est nécessaire de réaliser les bases du code Python permettant une première exécution de l'application Flask. Après l'import du module Flask et la création d'une variable représentant l'instance de l'application, nous pouvons attaquer le contenu du site.

Pour cela, Flask donne la possibilité de définir des pages individuelles associées à des routes, autrement dit des URLs*. On commence par créer une fonction initiale qui sera associée à la racine du site, c'est-à-dire à l'adresse « / ». De la même manière qu'une fonction Python standard, le choix nous appartient de mettre à l'intérieur les opérations que l'on veut effectuer au moment de l'accès au site. Par la suite, on peut également définir de nouvelles fonctions de routage associées à de nouvelles pages. Pour différencier ces fonctions de celles qui effectuent des traitements de fond sans affichage, on doit apposer au début de la fonction un décorateur. Celui-ci l'enveloppe et étend son comportement. Certains décorateurs prennent des arguments que l'on doit spécifier et c'est le cas du décorateur de routage, qui prend en option l'URL* choisi par le développeur pour la page. Il est possible ensuite d'ajouter du comportement et des interactions au sein des fonctions, par exemple générer une page graphique ou encore rediriger vers une autre page ou URL*. Voici un exemple simple de ce type de fonction Flask :

```
#### Home page rooting function ####
@app.route("/home")
def home():
    initCams()
    if is_logged_in:
        return render_template("home.html",
                               title="Home",
                               activetab='home',
                               cameras = cameras)
    else:
        return redirect(url_for('login'))
```

Figure 17 : Exemple de fonction de routage Flask

Vous pouvez voir ici le décorateur « @app.route » permettant de définir une fonction de routage vers l'URL* « /home » indiqué. Au sein de cette fonction, on exécute d'abord une fonction de traitement,

puis on contrôle la variable globale au site « `is_logged_in` », et en fonction de si l'utilisateur est connecté, on charge une page graphique avec la fonction Flask « `render_template` » ou bien on redirige vers une autre page avec « `redirect` ».

Un des avantages d'utiliser Flask pour créer son propre site web, est qu'il est possible pour un petit projet d'avoir un seul fichier Python pour le back-end entier de son site, tout en gardant un code clair et lisible. Celui-ci peut donc contenir les variables de configuration ou de traitements, les fonctions de traitements, de routage ou autres, ainsi que certaines données statiques. Dans le développement de ce projet, j'ai appliqué cela car je n'ai qu'un seul script nommé `app.py`. Toujours dans cette optique et contrairement à ce qui a été dit avant, j'ai par la suite ajouté au sein de ce même fichier toute la logique de fonctionnement du serveur Cloud ZMQ, j'explique les raisons dans la partie dédiée à l'infrastructure ZMQ.

3.1.2 Front-End

En parallèle du développement back-end, la création de la partie front-end permet de donner forme à l'interface du site, ce que voient les utilisateurs et avec quoi ils interagissent. Composé dans sa forme la plus simple par des fichiers HTML pour la structure et des fichiers CSS pour le style et la personnalisation, il est cependant possible d'opter pour des bibliothèques d'outils et composants complexes pré-sélectionnés. Pour ma part j'ai utilisé Bootstrap, une bibliothèque HTML, CSS et JavaScript, proposant une multitude de composants préconstruits et faciles d'intégration et personnalisation.

Comme détaillé dans le maquettage de l'application, nous avons 4 pages et donc logiquement un fichier HTML pour chaque. Ce sont ces fichiers que l'on va appeler « templates » et qui seront utilisés dans la fonction « `render_template` » pour générer l'interface graphique d'une page dans le code Flask Python.

La page de connexion possède son propre fichier HTML ainsi que sa propre fiche de style CSS. La barre de navigation située en haut de la fenêtre est en réalité un composant Bootstrap appelé NavBar. Ce composant, comme la plupart des autres composants de la bibliothèque, utilise des balises standards du langage HTML et se sert de la propriété « `class` » pour ajouter du style et du comportement. Le formulaire de connexion, élément principal de cette page, est quant à lui constitué d'un empilement de balises de divisions, utiles pour la mise en forme et la création de l'effet flottant et centré. A l'intérieur de ces divisions on trouve le formulaire composé de deux champs de saisie et d'un bouton de validation. Pour assurer le fonctionnement avec le back-end, j'ai créé une classe « `LoginForm` » qui utilise le module `FlaskForm`. Cette classe effectue le lien avec l'interface graphique et fournit les textes à afficher dans les champs de saisie ainsi que les fonctions de validation et de vérification des informations saisies.

Une fois l'utilisateur connecté, les différentes pages et onglets peuvent être consultées. Étant donné que chaque page du site, conformément au maquettage, possède la même barre de navigation et que le seul le contenu affiché change, il faut réutiliser ce composant sur les différents fichiers templates HTML. Cependant pour éviter la duplication inutile de code, j'ai utilisé le Template Designer de Jinja et plus précisément sa fonction d'héritage de templates. Cette fonction permet simplement de réutiliser des morceaux de code entiers de structure HTML dans d'autres fichiers sans réécrire le code en question. J'ai

donc créé un fichier de « layout » assurant la mise en page commune à toutes les pages du site à l'exception de la page de connexion qui gardera des fichiers séparés par soucis de sécurité et d'accès aux données, notamment par des injections* HTML.

Comme vous pouvez le voir sur le schéma ci-dessous, la structure de la mise en page est composée du header puis du corps de la page dans lequel on retrouve toute la barre de navigation et ses onglets et enfin le bloc de contenu de la page.

Mise en page HTML (Layout)

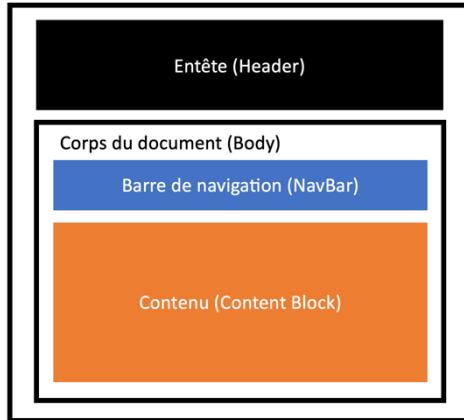


Figure 18 : Structure mise en page HTML du site

Constituer la mise en page comme ceci ne restreint pas la personnalisation car via la génération des pages avec la fonction Python « render_template », il est possible d'envoyer des données à travers certaines variables. En effet, Jinja permet d'effectuer des tests de conditions au sein des fichiers HTML. Cela m'a notamment servi afin de changer le titre de la page en cours de visualisation, chose qui se réalise seulement dans le header. Il nous suffit de créer une variable dans le code Python Flask, de transmettre la donnée à la génération de la page et la traiter au sein de la page HTML :

Flask Python	HTML Layout
<pre>return render_template("home.html", title="Home", activetab='home', cameras = cameras)</pre>	<pre><!-- Page Title --> {% if title %} <title>UiT Robot Surveillance - {{ title }}</title> {% else %} <title>Robot Surveillance</title> {% endif %}</pre>

Figure 19 : Relation entre Flask Python et code HTML Jinja

Chaque page du site a un contenu différent, on doit donc écrire le code de l'interface au cas par cas. Pour cela, on spécifie dans le layout un bloc de code de contenu avec la syntaxe « `{% block content %}{% endblock %}` » bloc qui sera définissable dans tous les fichiers HTML qui hériteront de ce layout. Par la suite, on hérite de la mise en page commune et on définit le contenu de la page comme ceci :

```
<!-- ----- -->
<!-- Page exemple : utilisation du layout et définition du contenu -->
<!-- ----- -->

<!-- On récupère le code commun de mise en page (Header, Navbar, etc...) -->
{% extends "layout.html" %}

<!-- On définit le contenu spécifique à la page -->
{% block content %}
    <div class="contenu">
        | <!-- Contenu -->
    </div>
{% endblock content %}
```

Figure 20 : Structure HTML d'une page de contenu

Ainsi, toutes les pages du site web bénéficieront du même entête de page personnalisé en fonction, de la même barre de navigation dynamique, des mêmes ajouts de scripts et d'autres éléments.

3.1.3 Master Detail

Je vais maintenant m'attarder sur le développement du contenu de la page de visualisation détaillée des caméras car il utilise des fonctions intéressantes de tous les langages, Frameworks* et bibliothèques utilisées. C'est également le contenu qui m'a pris le plus de temps à développer, m'ayant posé quelques soucis.

Comme expliqué précédemment, le contenu de cette page est agencé sous la forme d'un master-detail, c'est-à-dire une liste maître d'éléments et un affichage détaillé de l'élément sélectionné. La liste des caméras est contenue dans un composant Bootstrap nommé « Sidebar » et sera donc situé dans le premier tiers gauche de l'affichage. Plutôt que de coder chaque élément représentant les caméras en dupliquant à chaque fois le code comme je l'ai fait au début, j'ai finalement opté une nouvelle fois pour le Template Designer de Jinja et plus particulièrement la fonction de génération « for ».

Pour l'utiliser, il suffit de :

- Créer une liste de données de tous les éléments de la liste maître dans le code Python de l'application Flask.
- Transmettre ces données via la génération du template de la page du site (fonction « `render_template` »)
- Créer une structure HTML permettant l'affichage d'un élément de la liste.
- Mettre cette structure dans une boucle for Jinja et l'adapter en fonction des données reçues.

Utiliser ce principe est préférable car il permet d'anticiper dès à présent les futurs ajouts de caméras dans les données statiques du site. Aucune modification du code HTML ne sera donc nécessaire car l'affichage dynamique se mettra automatiquement à jour. Voici un exemple de code correspondant :

```
<div id="sidebarMenu>
  {% for cam in cameras %}
    <div>
      <a
        class="list-group-item list-group-item-action py-2
        aria-current="true"
        data-cam='{{ cam | toJSON | replace("\\"", "\\"") }}'
      >
        {{ cam.name }}
        {% if cam.status=='active' %}
          <span class="dot cam-active"></span>
        {% else %}
          <span class="dot cam-inactive"></span>
        {% endif %}
      </a>
    </div>
  {% endfor %}
</div>
```

Figure 21 : Génération liste dynamique avec Jinja

Une fois un élément de la liste sélectionné, il faut afficher dans l'espace restant les détails de la caméra ainsi que le flux vidéo. J'ai donc créé une nouvelle structure HTML simple pour cet affichage composée d'une zone de texte pour le nom de la caméra/angle de vue, un rond de couleur affiché à coté pour signifier le statut de la caméra et enfin la zone de vidéo.

Pour effectuer le lien entre le clic de l'utilisateur sur un élément de la liste et la mise à jour de l'affichage du détail, j'ai dû créer un script JavaScript associé à la page. Après de multiples tentatives infructueuses, la meilleure solution a été de récupérer les différentes balises HTML de la structure via leur attribut « class » ou « id » afin de détecter les changements d'états de la liste et de changer les propriétés de contenu du détail. Pour cela, après avoir récupéré sous forme de variables tous les éléments graphiques, j'ajoute à la liste maître un observateur d'événement de type « click ». Quand un des éléments de la liste sera cliqué, cet observateur s'activera et renverra une variable contenant notamment l'élément sélectionné. Après avoir récupéré les données caractéristiques de la caméra sélectionnée, j'effectue dans la suite logique les changements sur les éléments d'interface. Je remplace le texte du nom de l'angle de vue, je vérifie à nouveau le statut et je change la couleur en fonction. Enfin je change la source de la zone vidéo pour afficher le flux qui nous intéresse.

Lors de l'accès à la page de visualisation détaillée via les onglets de la barre de navigation, aucune caméra par défaut n'est sélectionnée. Au lieu de charger la page en affichant une zone de détail vide, j'ai décidé de créer une structure alternative affichant seulement un texte de guide pour l'utilisateur. Dans le comportement du script JavaScript, quand aucun élément n'est sélectionné par défaut, cette structure est affichée en remplacement.

3.2 Serveur-client ZMQ

La deuxième partie importante de mon stage a été de développer les fondements de l'infrastructure réseau assurant le fonctionnement de la vidéo-surveillance. Je me suis donc dirigé vers la création de deux serveurs ZMQ : un local et un distant.

3.2.1 Version de démonstration

Pour commencer ce développement, on m'a demandé de réaliser une première version de démonstration de ce type d'infrastructure à mettre en œuvre en restant pour le moment sur une même machine. Le but étant de réussir à connecter les deux serveurs pour pouvoir afficher les flux vidéo depuis le client.

3.2.1.1 Structure

Le fonctionnement de ZMQ est basé sur une API* utilisant les sockets réseaux. Ce sont des interfaces de connexion réseau logicielles présentes dans la plupart des systèmes d'exploitation. En pratique, le nom socket désigne en réalité une variable employée dans un programme afin de gérer les sessions de connexion réseau. Ce qui rend les sockets ZMQ particulièrement intéressantes, est qu'elles offrent une meilleure abstraction des protocoles réseau sous-jacent mais aussi parce qu'il existe différents types de sockets pour développer de multiples modèles de messagerie, qui sont implémentés par des paires de sockets avec des types correspondants. Certains modèles sont encore à l'état d'expérimentation mais les modèles intégrés sont :

- Le modèle demande-réponse (REQ/REP) basé sur un appel de procédure à distance. C'est un modèle de distribution de tâches.
- Le modèle publieur-abonné (PUB/SUB) qui connecte un ensemble de publieurs à un ensemble d'abonnés. Il s'agit ici d'un modèle de distribution de données, donc plus adapté à ce projet.
- Le modèle Pipeline qui est un modèle de distribution et de collecte de tâches parallèles.
- Et enfin le modèle de pair exclusive, qui relie exclusivement deux sockets. Cependant il s'agit ici d'un modèle pour connecter deux sous-tâches d'un processus.

Comme mentionné précédemment, la première idée pour ce projet a été d'utiliser le modèle PUB/SUB car cela semble le plus adapté pour le projet. A l'image du schéma ci-dessous, le publieur est en réalité le serveur local chargé de récupérer les flux depuis les caméras, l'abonné étant le serveur distant.

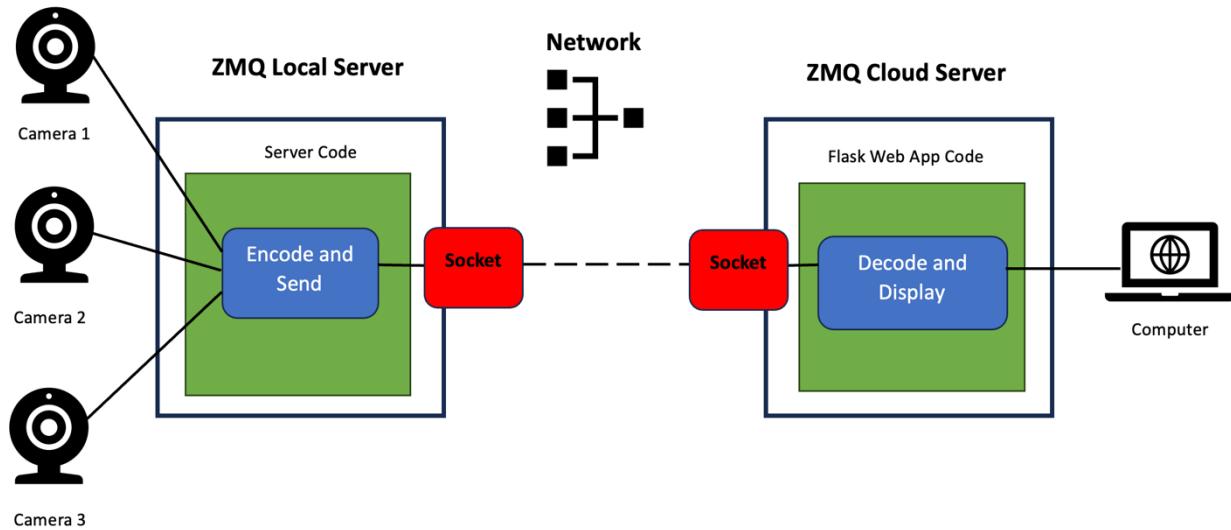


Figure 22 : Schéma représentant l'infrastructure réseau ZMQ de démonstration

Pour appliquer ce modèle de communication dans le code des serveurs il faut commencer par créer les sockets. On déclare alors une socket de type PUB dans le serveur local et une autre de type SUB dans le serveur Cloud. Par la suite, pour mettre en relation ces deux sockets, il existe deux fonctions : « bind » et « connect ». Il faut savoir que ZMQ crée des files d'attentes pour chaque connexion sous-jacente. Ainsi avec « bind », on autorise les pairs à se connecter. On ne saura donc pas combien il y en aura et donc on ne peut pas créer les files d'attentes à l'avance. En revanche, avec « connect », on sait qu'il y aura au moins une seule connexion, ce qui nous permet de créer automatiquement et immédiatement la file d'attente. Ce qui explique que dans la plupart des cas, le serveur utilise « bind » pour attacher des sockets à des ports de diffusion, tandis que le client utilise « connect » pour s'abonner à un seul et même diffuseur. Dans mon cas et comme vous pouvez le voir sur l'image ci-dessous, le serveur local attache sa socket au port local n°5555 qui communiquera en TCP*. Le serveur distant connectera donc sa socket à l'adresse de diffusion précédente qui, dans le cadre de la version de démonstration, reste sur la même machine et est donc « localhost ».

```
# Serveur local
context = zmq.Context()
socket = context.socket(zmq.PUB)
socket1.bind("tcp://*:5555")

# Serveur distant
context = zmq.Context()
socket = context.socket(zmq.SUB)
socket1.connect("tcp://localhost:5555")
```

Figure 23 : Connexion serveurs ZMQ publieur/abonné

Étant donné que plusieurs types d'information différents peuvent transiter à travers ses sockets, j'ai jugé important d'appliquer le mécanisme de « **topic** » ou autrement dit sujet. A l'envoi de données

dans le serveur local, on ajoute un petit code texte sous forme de chaîne de caractère qui agira comme une catégorie de donnée. La socket réceptrice du serveur distant doit souscrire à ce topic afin de filtrer les données reçues et ne garder que ce qui l'intéresse. On utilise pour cela la fonction « subscribe ».

3.2.1.2 Traitement vidéo

Une fois la connexion entre les pairs établie, il faut maintenant traiter et communiquer l'information vidéo. Pour cela on va utiliser des fonctions très utiles de la librairie OpenCV2.

Les caméras mises à ma disposition possèdent toutes une interface graphique de configuration sur laquelle on peut notamment paramétriser la qualité, la fluidité ou l'encodage du flux vidéo. Chaque flux émis par les caméras est accessible via une URL* utilisant le protocole RTSP* comme : « rtsp://10.0.0.30:554/stream1 ».

Le serveur local commence donc par capturer les flux vidéo grâce à ces URL*, en utilisant la fonction « VideoCapture ». Par la suite, pour traiter cette vidéo afin de la transmettre au serveur distant, il faut réaliser différentes opérations image par image et donc frame par frame. Pour récupérer une frame on utilise la méthode « read » de la capture vidéo effectuée précédemment. Après avoir obtenu l'image, on peut effectuer les traitements suivants : on encode l'image en format jpg pour plus de compatibilité puis on transforme cette image jpg en séquence de bytes pour transmission. Enfin on peut maintenant envoyer une trame de données au serveur distant en concaténant un topic prédéfini et la séquence de bytes représentant l'image via la fonction « send_multipart » de la socket ZMQ. Une vidéo étant une série continue d'images nécessitant toutes ce traitement, on englobe ces opérations dans une boucle. Voici ci-dessous pour plus de clarté le code correspondant :

```
# On définit le topic
topic = 'cam1'
# Capture vidéo
cap = cv2.VideoCapture('rtsp://10.0.0.30:554/stream1')

# Boucle de traitement et d'envoi
while True:
    # On lit la frame
    ret, frame = cap.read()
    # On traite la frame
    encoded_frame = cv2.imencode('.jpg', frame)
    frame_data = encoded_frame[1].tobytes()
    # On envoie la frame
    socket.send_multipart([topic.encode(), frame_data])
```

Figure 24 : Récupération, traitement et envoi vidéo

Maintenant, après réception de données dans le serveur distant, et en fonction du topic reçu, on effectue les transformations inverses en récupérant la séquence de bytes et en l'encodant sous forme d'image jpg. Suivant la méthode d'affichage utilisée, on peut être amené à transformer encore ces données pour qu'elles soient davantage exploitables.

3.2.2 Implémentation locale

Une fois l'infrastructure de démonstration opérationnelle, il a fallu implémenter ce mécanisme dans la globalité du projet de vidéo-surveillance. Dans un souci de simplicité, j'ai préféré fusionner directement le code du serveur distant avec le code du site Flask, étant donné que cela ne pose aucun problème lors de l'exécution. La première étape de l'implémentation a continué à se faire en réseau local mais cette fois sur deux machines différentes. C'est à ce moment-là que des problèmes relativement gênants sont apparus, dont certains m'ont forcé à revoir mon architecture.

3.2.2.1 Multithreading

Le premier problème rencontré lors de cette implémentation a été une latence vidéo élevée et exponentielle dans le temps. En effet, le délai d'affichage de la vidéo de chaque caméra était au lancement de l'application d'environ 1 à 2 secondes, ce qui reste correct mais cela augmentait très vite et de façon exponentielle, pouvant atteindre au bout de 30 secondes de temps écoulé jusqu'à 30 à 45 secondes de latence. Exposant le problème à mes tuteurs, on m'a conseillé de me documenter sur le multithreading.

Le multithreading étend l'idée du multitâche aux applications, c'est-à-dire de pouvoir diviser certaines opérations spécifiques de la même application en threads individuels pouvant fonctionner en parallèle. Cela permet dans de nombreux cas d'accélérer la réactivité de l'application et de tirer parti du matériel sur lequel elle s'exécute, si c'est un système multiprocesseur ou multicœur, globalement répandus de nos jours.

Il existe dans les bibliothèques standards Python, un module nommé « `threading` » permettant justement d'appliquer du parallélisme basé sur les fils d'exécution et donc les threads. Appliquer son fonctionnement au projet a été relativement simple. Il a fallu créer un thread par caméra, chargé d'exécuter en continu et en parallèle les différents traitements sur les images puis de les envoyer. Le code à effectuer par les threads, doit obligatoirement être disposé dans une fonction Python. J'ai ainsi créé une fonction « `capture_send_video` » prenant en arguments une socket et une capture vidéo. Celle-ci contient la boucle de traitement et d'envoi mentionnée précédemment.

Dans un premier temps, cela a permis de réduire grandement l'augmentation exponentielle du délai d'affichage mais il était toujours présent. Il devait y avoir un goulot d'étranglement quelque part et celui-ci n'était autre que la socket ZMQ. Limitée par son débit de traitement mais également par le débit du réseau, cela continuait de créer une latence à l'affichage car les 3 flux vidéo transitaient à travers. J'ai donc dû créer par la suite de nouvelles sockets aussi bien du côté diffuseur qu'abonné, afin que chacune ne soit dédiée qu'à une seule et même caméra.

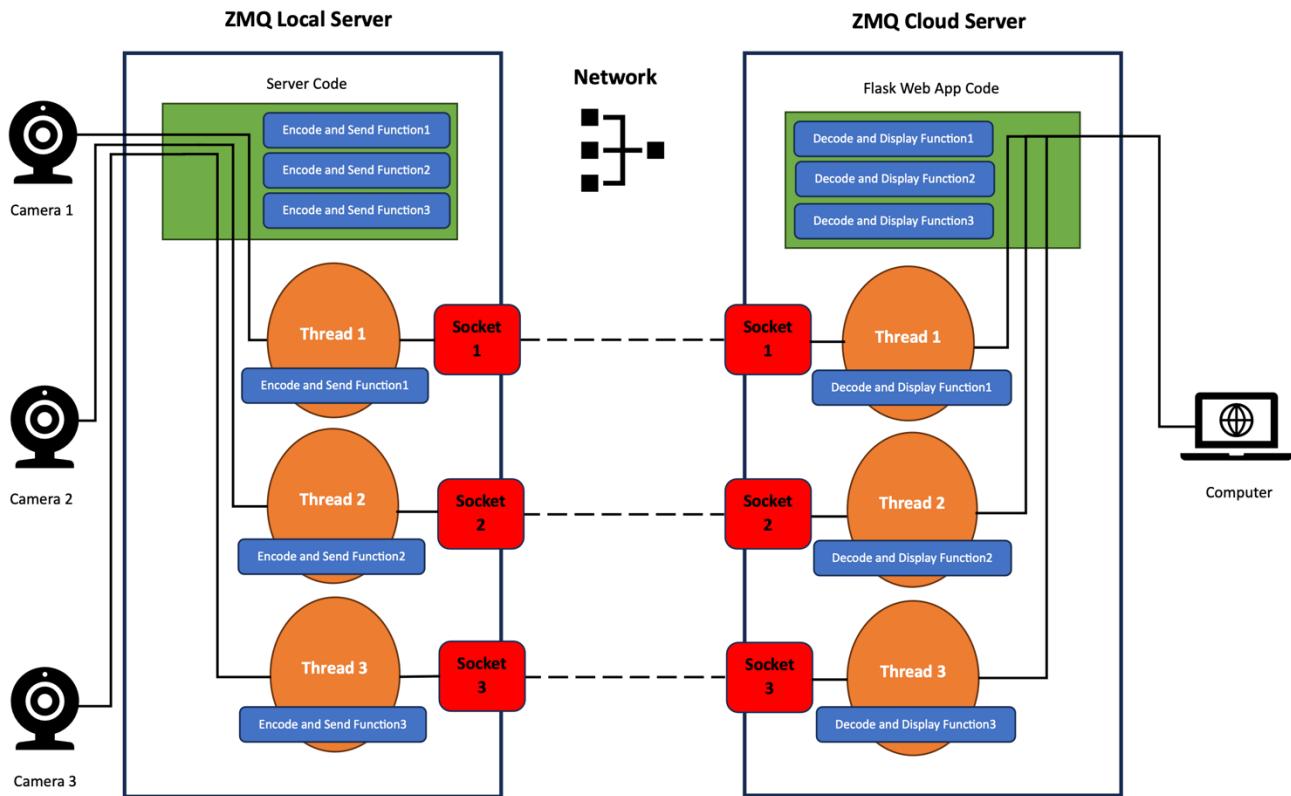


Figure 25 : Schéma infrastructure multi-threadée

Comme vous pouvez le voir sur l'image ci-dessus, j'ai également appliqué le multithreading à l'abonné et donc directement dans le serveur ZMQ Cloud. Cela a aussi contribué à fluidifier davantage le traitement des données reçues ainsi que l'affichage. De la même manière que pour le serveur diffuseur, j'ai créé la fonction « receive_encode_video » qui englobe la réception des données, les opérations d'encodage ainsi que l'envoi pour affichage à la page HTML. Comme vous pouvez le voir dans l'exemple de code ci-dessous, c'est cette fonction qui sera exécutée en continue dans les threads côté site web.

```

import threading

# Création du thread
thread = threading.Thread(target=fonction_a_executer, args=(socket, cap))

# Exécution du thread
thread.start()

# Attente de la fin d'exécution du thread
thread.join()

```

Figure 26 : Exemple de manipulation des threads en Python

3.2.2.2 Rattrapage des images

Le second et important problème que j'ai rencontré a été observé lorsque j'ai décidé d'ajouter l'affichage de la date et l'heure directement au sein des flux vidéo, grâce à l'interface de configuration des caméras. J'ai alors remarqué dans la page de visualisation détaillée, que lorsque je quittais la vue d'une caméra pour une autre en cliquant dans le master-detail, et que je revenais ensuite dessus, le temps affiché sur la vidéo restait identique. J'avais donc un retard temporel dans la visualisation qui dépendait du temps de non-visualisation. L'affichage ne se mettait pas à jour en temps réel, ce qui est contraire au principe de vidéo-surveillance en temps réel.

J'ai mis longtemps à trouver la source du problème. En effet j'ai d'abord pensé que le problème provenait du programme interne des caméras. Après visualisation des flux en utilisant le logiciel VLC, il n'y avait aucun problème de ce côté-là. Ensuite j'ai revu en détail tout mon code. J'ai identifié deux sources potentielles ; les threads et les sockets.

J'ai pensé en premier lieu que le problème venait des threads fraîchement ajoutés, qui pour une raison inconnue ne fonctionneraient pas correctement. Après beaucoup de débogage, je n'ai ni trouvé de problèmes à l'exécution des threads ni par rapport à leur communication avec les sockets. J'ai donc par la suite réalisé une inspection complète des sockets ZMQ. En inspectant les logs d'exécution de l'application, j'en ai déduit que, étant donné que le serveur local d'envoi fonctionne en continu, une file d'images en attente d'être traitées et affichées se créait dans la socket. Après des recherches sur internet, j'ai compris qu'il était possible de « flush » les sockets et donc de vider la file d'attente de messages reçus. Cependant, je n'ai trouvé aucune façon de réaliser cette opération avec les sockets ZMQ. Il a donc été nécessaire de créer, uniquement au sein du serveur receveur, un mécanisme de rattrapages et saut des images perdues afin de revenir en visualisation direct. Pour cela, j'ai dû enregistrer dans une variable le dernier temps de visualisation du flux vidéo, faire une différence avec le temps actuel lors de la reprise de la visualisation et calculer ainsi le nombre d'images à rattraper. Le rattrapage des frames s'effectue simplement en recevant en boucle les images manquées sans les traiter. Voici le code correspondant :

```
# Skipping lost frames
current_time = time.time()
if last_visualization_time is not None:
    elapsed_time = current_time - last_visualization_time
    skipped_frames = int(elapsed_time / frame_time)
    for _ in range(skipped_frames):
        try:
            socket.recv_multipart(zmq.NOBLOCK)
        except zmq.error.Again:
            break
last_visualization_time = current_time
```

Figure 27 : Mécanisme de rattrapages des images perdues

Une fois ce mécanisme implémenté, j'ai toute de suite vu la différence. Le changement d'affichage se fait de manière fluide et sans temps de chargement. De plus, de retour sur un des angles de vue, les images manquées défilent rapidement avant de revenir à la vidéo en direct.

3.2.3 Implémentation distante

L'étape finale de ce stage a été de déployer une version fonctionnelle et stable cette infrastructure sur le Cloud. L'environnement d'exécution utilisé auparavant étant seulement un environnement de développement et non de production, il a été indispensable de trouver un nouvel outil destiné à cet usage. La documentation officielle de Flask nous recommande d'utiliser Waitress comme outil de production, j'ai donc opté pour celui-ci.

Waitress est serveur Python standardisé WSGI, spécifiant comment un serveur web peut interagir avec une application Python. Il n'a pas de dépendances autre que les bibliothèques standards Python et a des performances très acceptables. De plus, sa configuration est très simple car elle tient en une ligne à écrire dans la condition d'exécution du script Python :

```
if __name__ == '__main__':
    # Configuration du serveur de production Waitress
    serve(app, host='0.0.0.0', port=8080, threads = 6)
```

Figure 28 : Configuration du serveur Waitress

De la même manière que dans l'image ci-dessus, la fonction « serve » prend en paramètre la variable représentant l'instance de l'application (ici app), une URL* de déploiement (ici 0.0.0.0 signifie qu'on adopte l'adresse IP de la machine comme URL*), un port de connexion à l'URL* (ici 8080) et optionnellement le nombre de threads dédiés à la gestion des requêtes sur le site.

Ayant développé la plupart du projet en local, il a aussi été indispensable de changer les adresses de connexion des sockets. La machine virtuelle Cloud sur laquelle le site est déployé possède une adresse IP publique mais ce n'est pas le cas du serveur local, intégré au sein du réseau de l'université, possédant donc une adresse privée. L'architecture selon le modèle PUB/SUB ne convenait donc pas dans ce sens-là. J'ai donc cherché sur la documentation Flask ainsi que sur d'autres sites internet comment remédier à ce problème. La meilleure solution trouvée a été de faire passer cette architecture en modèle Pipeline et donc avec des sockets PUSH/PULL, en adaptant le tout pour convenir à des données plutôt qu'à une distribution de tâches. Cette fois la socket de type PUSH distribuera à sa socket cliente de type PULL. L'adressage et la connexion se font donc dans le sens inverse : la socket émettrice se connecte directement à l'adresse publique du serveur distant via la fonction « connect » et la socket réceptrice utilise la fonction « bind » pour écouter sur son interface réseau en attente des données.

3.3 Résultats et améliorations

Dès l'instant où le déploiement de l'infrastructure serveur et du site web de vidéo-surveillance a été réalisé, j'ai entamé une courte phase de peaufinage, ayant pour but d'améliorer le résultat de mon travail.

3.3.1 Résultats

Concernant les résultats que j'ai obtenus à l'issu de la réalisation de ce projet, je me suis donné comme objectif de rester le plus fidèle possible au maquettage réalisé pour fournir une expérience complète, intuitive et stable.

Un utilisateur souhaitant accéder au site débouche premièrement sur la page de connexion que voici :

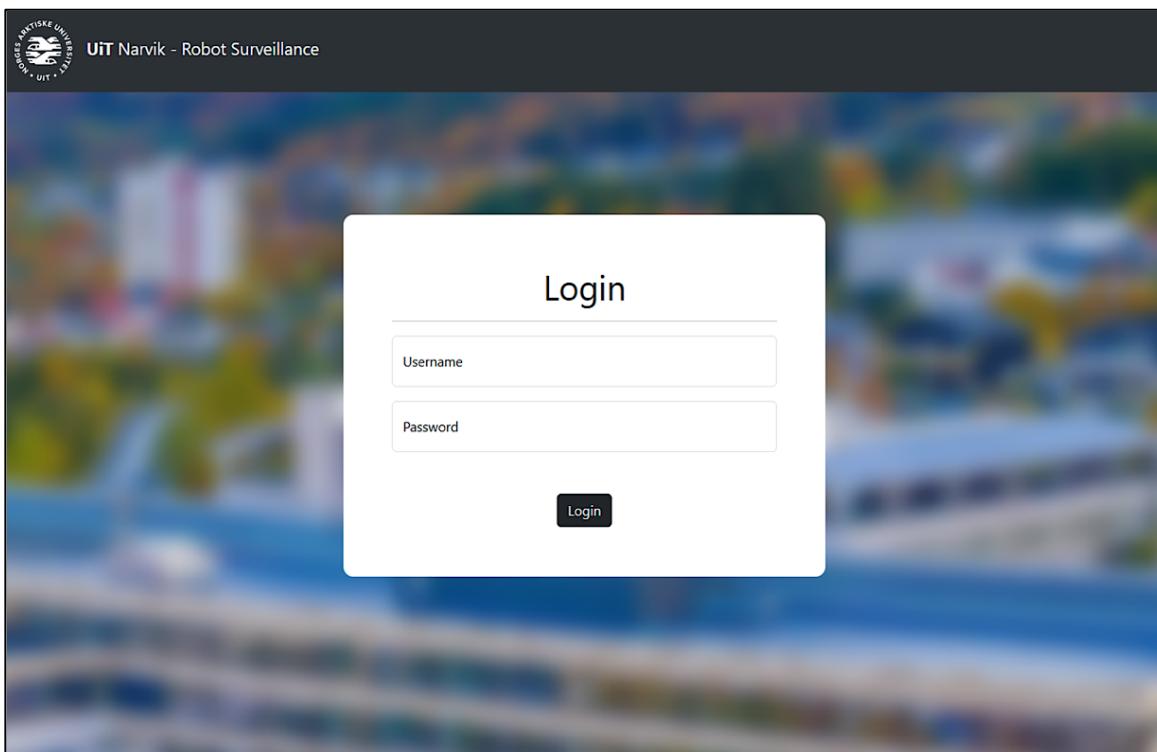


Figure 29 : Page de connexion du site

Une fois connecté, l'utilisateur arrive sur la page d'accueil ou il peut naviguer via les onglets de la barre de navigation, visualiser grâce au graphique l'emplacement de ces caméras dans la zone de manipulation du robot assistant et enfin cliquer sur les boutons pour accéder directement à la visualisation de ces caméras :

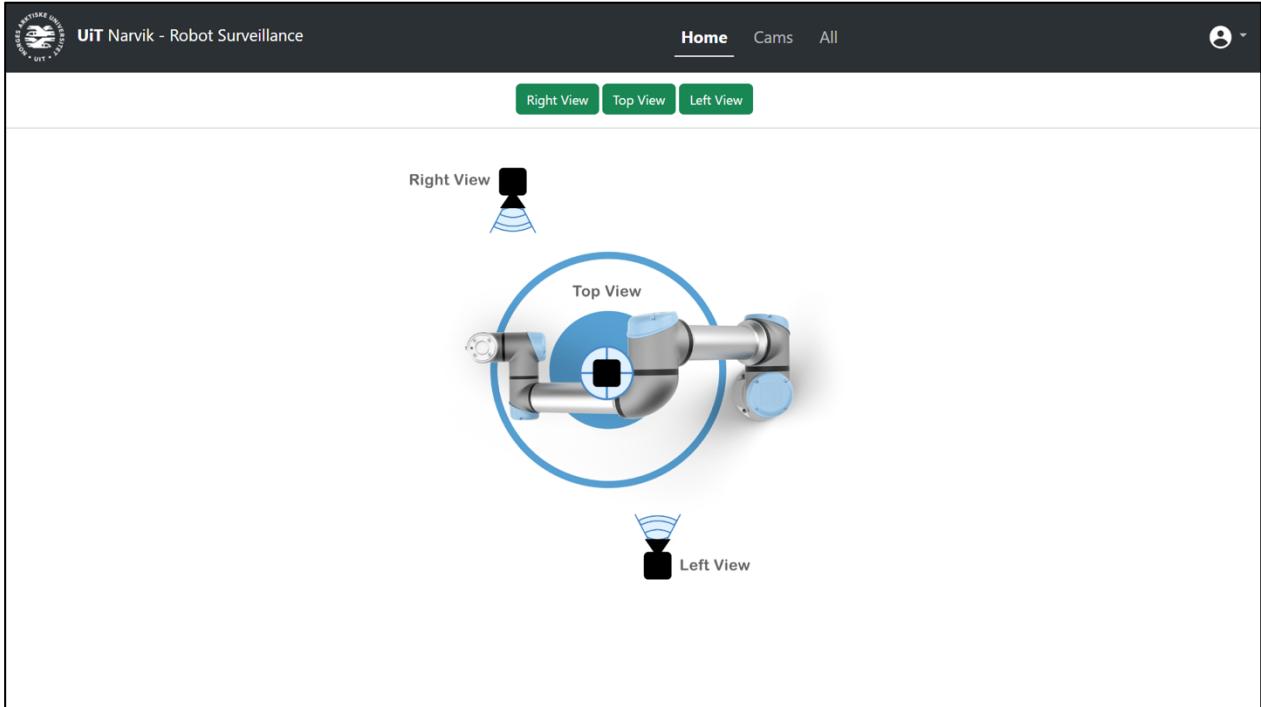


Figure 30 : Page d'accueil du site

Lorsque l'utilisateur se rend sur le deuxième onglet du site nommé « cams », il est affiché le master-detail lui permettant de visualiser en premier plan une caméra et changer de point de vue au sein de la même page :

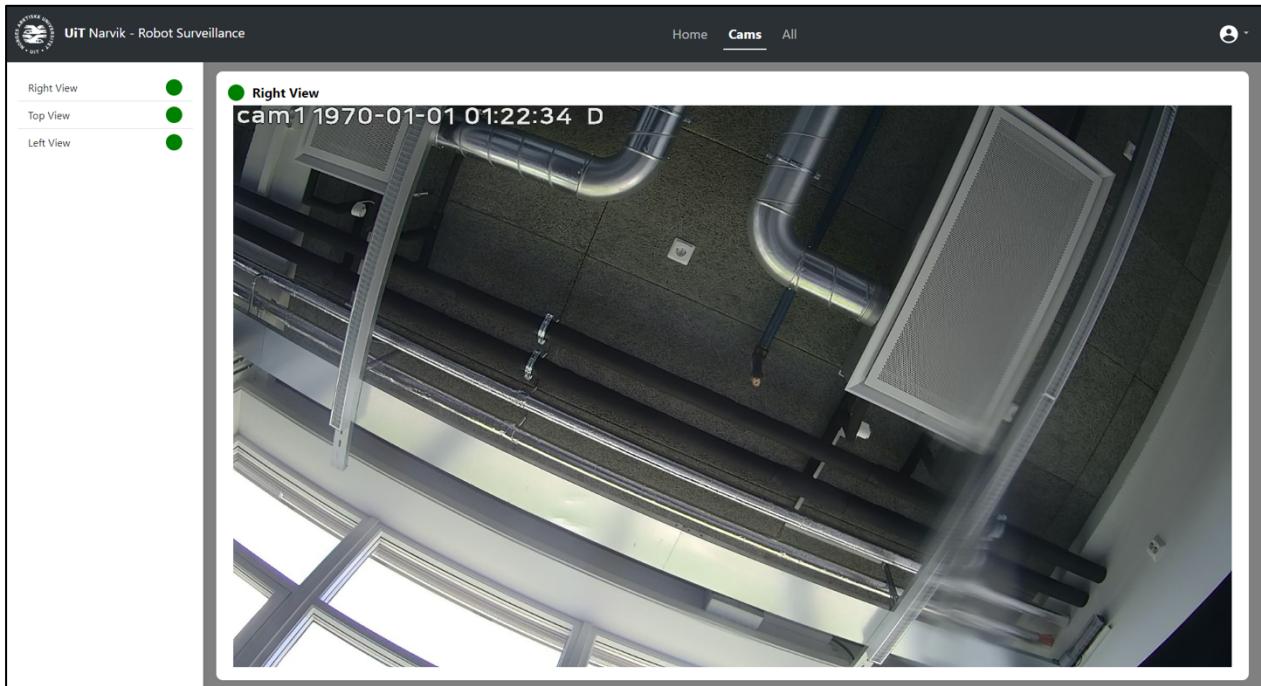


Figure 31 : Page de visualisation détaillée du site

Enfin, le dernier onglet du site permet à l'utilisateur d'avoir une vue globale de toutes les caméras ainsi qu'un rappel du schéma permettant de visualiser l'emplacement des celles-ci dans la zone du robot :

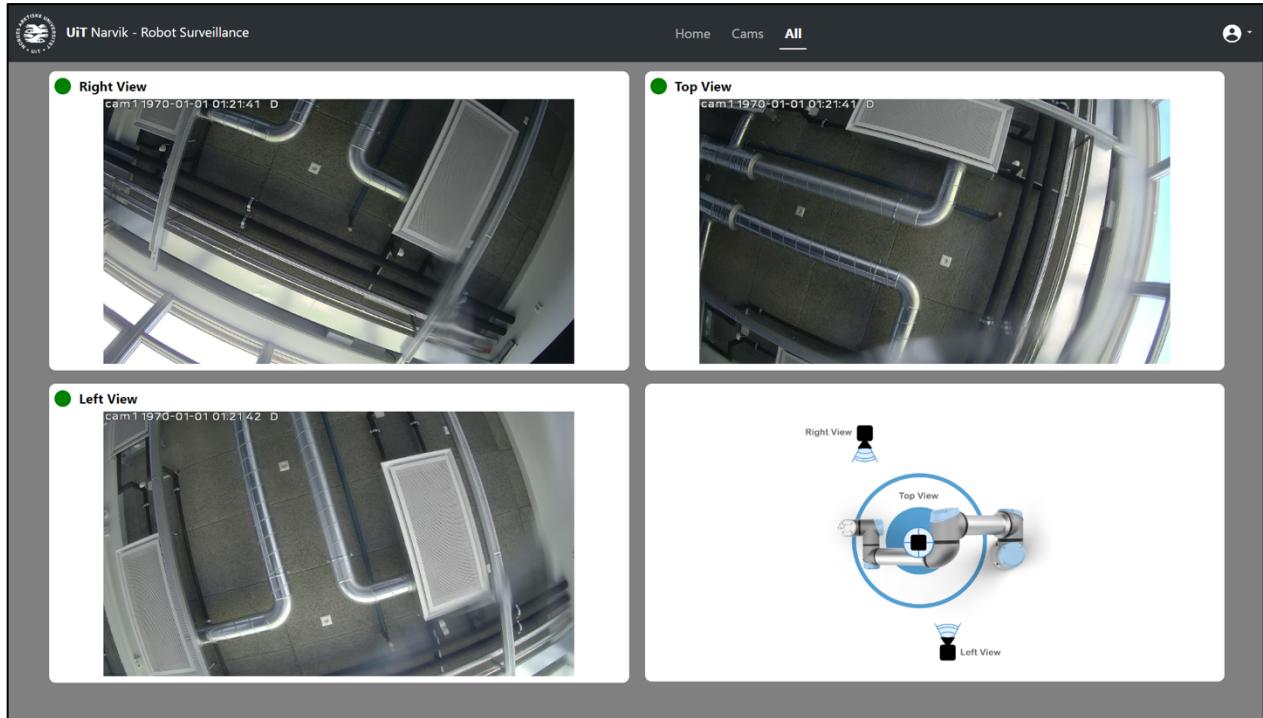


Figure 32 : Page de visualisation globale du site

Pour ce qui est de l'infrastructure réseau développée, aucun résultat visuel n'est à présenter mais le code fonctionnel est disponible sur ma page GitHub :

<https://github.com/louprusak/UiT-Narvik-Robot-Surveillance>

3.3.2 Améliorations

Les améliorations possibles pour ce projet de stage sont multiples. Compte tenu du contexte de recherche et d'autonomie dans lequel j'ai travaillé, j'ai découvert et expérimenté beaucoup de nouveaux aspects du développement web ainsi que de technologies. Bien entendu, ces améliorations concernent les deux parties distinctes de ce projet : le site web Flask de vidéo-surveillance et l'infrastructure réseau ZMQ.

D'abord, comme exposé dans la partie maquettage de l'application, je souhaitais proposer sur la page d'accueil, une image cliquable et dynamique affichant les caméras et leurs statuts. J'ai entamé le développement notamment en utilisant la balise « map » de HTML qui permet de disposer divers éléments avec un système de coordonnées mais cela ne s'adaptait pas bien en fonction des différents ratios d'écrans. Par soucis de temps et de priorité, je l'ai remplacé par une image et des boutons statiques.

Toujours concernant le site web et son interface graphique, certains éléments d'interface ne sont pas encore parfaitement « responsive », c'est-à-dire que si on redimensionne la page en petite taille ou bien que l'on change le ratio d'écran, certains éléments d'interface ne s'adaptent pas parfaitement. Actuellement, bien que ce ne soit pas un usage indispensable, la visualisation des flux vidéo depuis un smartphone n'est pas optimisée.

Cette fois-ci concernant l'infrastructure réseau, plusieurs améliorations sont envisageables :

- Séparer le code du serveur Cloud du code de l'application web Flask. Ce changement serait préférable étant donné que ce serveur est amené à évoluer et à accueillir d'autres fonctionnalités n'ayant pas forcément de lien avec la vidéo surveillance.
- Une meilleur gestion des erreurs dans le code des deux serveurs pour éviter les plantages inopinés afin que le processus reste en fonctionnement en continu.
- Une meilleure fonctionnalité de récupération du statut des caméras, car l'actuelle est lente à détecter qu'une caméra n'est pas connectée. Pour ces mêmes raisons de performance j'ai choisi de la simplifier pour le moment, ce qui peut entraîner des erreurs.

Beaucoup d'autres améliorations sont possibles, ce travail étant avant tout de la recherche, il sera forcément amené à être revu et amélioré par d'autres personnes au sein de l'université de Narvik.

Conclusion

Pour conclure, mon objectif durant ce stage a été d'effectuer un travail de recherche sur les technologies actuelles de développement web en Python, pour conceptualiser et réaliser une première solution de vidéo-surveillance répondant au mieux aux exigences formulées. Cette solution est composée d'une application web de visualisation créée avec Flask et également d'une infrastructure réseau assurant son fonctionnement, développée avec le Framework* ZMQ. Conformément aux attentes, la visualisation des flux vidéo en provenance des caméras est simple, intuitive et fluide.

Le développement de ce système a été complexe à mettre en œuvre car, travaillant en totale autonomie, j'ai parfois dû revoir quasiment intégralement la conception même de celui-ci, faisant face à différents problèmes de fonctionnement. C'est en faisant preuve d'initiative, de volonté et de persévérance tout au long de ce stage, qu'il m'a été possible de trouver des solutions ou des compromis à ces problèmes et de poursuivre mon travail dans les meilleures conditions.

Bien que dans l'état actuel, la solution est fonctionnelle et répond à la plupart des attentes, elle n'est pas exempte de défauts et elle pourrait nécessiter certaines améliorations. Il serait notamment intéressant de peaufiner le visuel du site et ses interactions afin qu'il soit adapté à plus de supports, mais également d'améliorer l'infrastructure réseau associée, afin d'accueillir dans le futur le standard OPCUA ainsi que les fonctionnalités d'autres machines industrielles du laboratoire.

En réalisant ce projet de recherche de A à Z, j'ai découvert une nouvelle manière de travailler et ai expérimenté de nombreux domaines du développement web, dont certains non envisagés. J'ai également beaucoup découvert et appris sur les technologies utilisées, dans leur fonctionnement mais surtout dans la manière de les utiliser et de les intégrer.

D'un point de vue personnel et en dehors du travail que j'ai réalisé, ce stage dans un pays étranger m'a beaucoup apporté. La découverte d'un nouveau pays, d'une nouvelle culture, d'une nouvelle manière de vivre ainsi que l'établissement de nouvelles relations sociales dans une langue différente de ma langue maternelle, m'ont fait sortir de ma zone de confort pour que cela soit finalement une merveilleuse expérience.

J'espère que mes réalisations auront su satisfaire mes tuteurs et responsables de stage, et qu'ils pourront les utiliser à leur convenance, les améliorer ainsi que les compléter, afin d'atteindre l'objectif fixé pour le laboratoire du département ingénierie industrielle de l'université de Narvik.

Le petit point environnement

Ayant voyagé, travaillé et vécu pendant 4 mois en Norvège, dans la petite ville de Narvik, située à 220km au nord du cercle polaire arctique, j'ai pu observer de nombreux paysages naturels. Située entre l'eau clair turquoise des fjords et les montagnes enneigées et escarpées, la ville est entourée de nature et les panoramas sont à couper le souffle. En effectuant des randonnées au bord des fjords, au cœur des forêts ou encore dans les montagnes et leurs nombreux lacs, on se rend compte de la beauté de cette nature et de la nécessité de la préserver. Bien que le climat soit quelque peu hostile et le rythme journalier chamboulé par de longues périodes de nuit ou de jour polaire, la faune et la flore est magnifique et extrêmement présente. On constate alors que, quel que soit notre situation géographique, associer progrès et respect de l'environnement n'est pas incompatible et changer certains petits éléments de nos habitudes de vie quotidienne suffirait à améliorer notre futur.

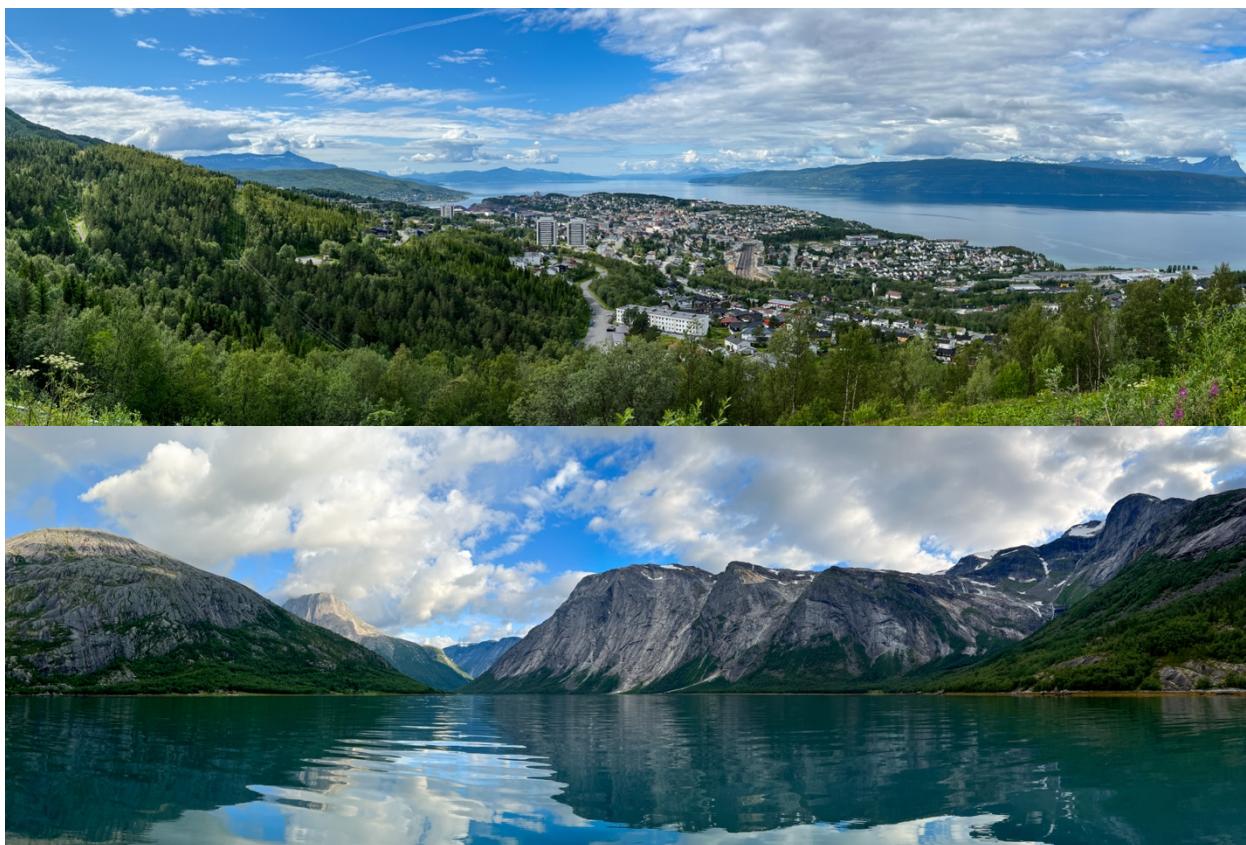


Figure 33 : Panoramas de la ville de Narvik et des fjords et montagnes alentours

Références bibliographiques

[1] Bootstrap Team, *Get started with Bootstrap*, Documentation Bootstrap, Juin 2020 [Online].
Disponible sur : <https://getbootstrap.com/docs/5.3/getting-started/introduction/>.
[Consulté le : 18-avril-2023]

[2] A. RONACHER, Flask Pallets Projects, *Flask User's Guide*, Documentation Flask 2.3, Mai 2023 [Online].
Disponible sur : <https://flask.palletsprojects.com/en/2.3.x/>.
[Consulté le : 17-avril-2023]

[3] A. RONACHER, Jinja Pallets Projects, *Documentation Jinja 3.1.2*, Avril 2022 [Online].
Disponible sur : <https://jinja.palletsprojects.com/en/3.1.x/>.
[Consulté le : 21-avril-2023]

[4] Intel, Willow Garage, *Documentation OpenCV 4.8*, Juin 2023 [Online].
Disponible sur : <https://docs.opencv.org>.
[Consulté le : 16-mai-2023]

[5] Python Software Foundation, *Documentation Python 3.11*, Juin 2023 [Online].
Disponible sur : <https://www.Python.org/doc/>.
[Consulté le : 10-avril-2023]

[6] Zope Foundation and Contributors, *Documentation Waitress 2.1.2*, Mai 2022 [Online].
Disponible sur : <https://pypi.org/project/waitress/>.
[Consulté le : 10-juil-2023]

[7] iMatix, *Documentation ZeroMQ Python*, Juillet 2019 [Online].
Disponible sur : <https://zeromq.org/get-started/?language=Python&library=pyzmq#>.
[Consulté le : 29-mai-2023]

Lexique

Framework : En programmation informatique, un Framework* est un ensemble cohérent de composants logiciels structurels qui sert à créer les fondations ainsi que les grandes lignes de tout ou partie d'un logiciel, c'est-à-dire une architecture.

Cobot : Robot assistant intelligent.

Industrie 4.0 : Synonyme de fabrication intelligente, l'industrie 4.0 est la concrétisation de la transformation numérique des opérations du domaine. Elle offre une prise de décision en temps réel, ainsi qu'une productivité, une flexibilité et une agilité accrues.

Fablab : Atelier mettant à la disposition du public des outils de fabrication d'objets assistée par ordinateur.

TCP : TCP/IP signifie Transmission Control Protocol/Internet Protocol (Protocole de contrôle des transmissions/Protocole Internet). TCP/IP est un ensemble de règles normalisées permettant aux ordinateurs de communiquer sur un réseau tel qu'Internet.

UDP : User Datagram Protocol est un protocole de communication de substitution à Transmission Control Protocol (TCP). Il est surtout utilisé pour établir des connexions à faible latence et à tolérance de perte entre applications sur Internet.

URL : Adresse d'un site ou d'une page hypertexte sur Internet.

Injection de code : Une injection de code est un type d'exploitation d'une faille de sécurité d'une application, non prévue par le système et pouvant compromettre sa sécurité, en modifiant son exécution. Certaines injections de code ont pour but d'obtenir une élévation des privilèges, ou d'installer un logiciel malveillant.

API : Une API (application programming interface ou « interface de programmation d'application ») est une interface logicielle qui permet de « connecter » un logiciel ou un service à un autre logiciel ou service afin d'échanger des données et des fonctionnalités.

RTSP : RTSP ou Real Time Streaming Protocol (protocole de streaming temps-réel) est un protocole de communication de niveau applicatif (niveau 7 du modèle OSI) destiné aux systèmes de streaming média.

Frame : image (en anglais).

Thread : Un thread est l'unité de base à laquelle un système d'exploitation alloue du temps processeur. Chaque thread a une priorité de planification et maintient un ensemble de structures utilisé par le système pour enregistrer le contexte du thread quand l'exécution du thread est en pause.