Projet simulateur IGOsat

Vladislav Fitz
Ulysse Gérard

IGOsat, un projet universitaire

- Concevoir un nano-satellite capable de réaliser des mesures et de les communiquer à la Terre.
- Conception et réalisation entièrement prise en charge par des étudiants.
- Principalement physiciens jusqu’à maintenant. Deux groupes d’étudiants en informatique cette année.
- Long-terme: lancement prévu pour 2018.

Un simulateur pour quoi faire ?

- Une simulation fonctionnelle et non matérielle.
- Permettre de dimensionner correctement le matériel. S’assurer de la justesse des flots de données entre les composants du satellite.
- Il s’agit d’une étude initiale grossière des besoins du projet. Cependant l’architecture du simulateur permettra dans le futur d’affiner la simulation.

- Générique et réutilisable dans d’autres projets du même type.

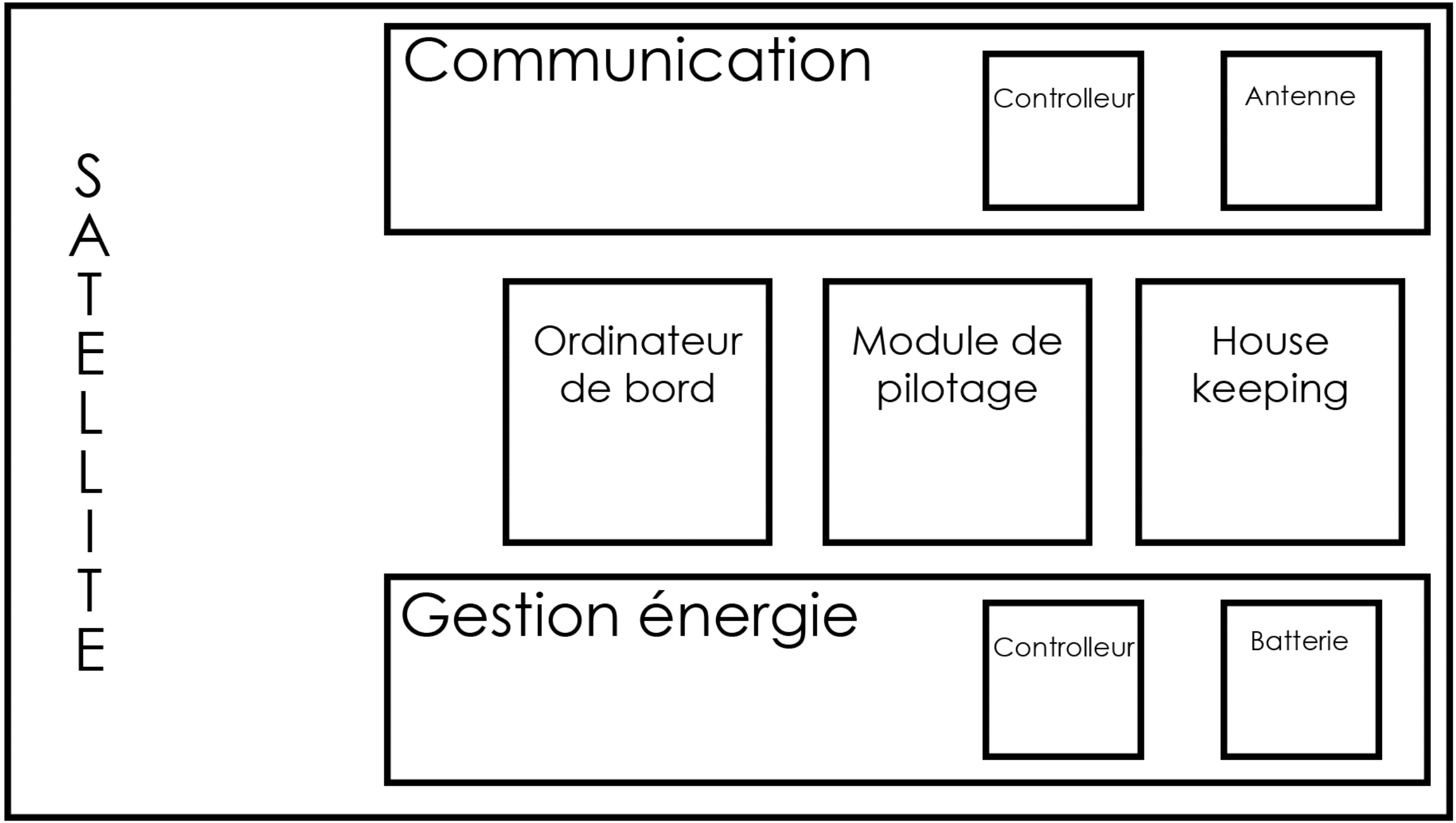
Un simulateur pour qui ?

- En l’état pas d’interface graphique ni d’outils de modélisation.
- S’adresse donc à des informaticiens capables de construire une instance du simulateur adaptée à leurs besoins autour du noyau abstrait.

- Une évolution future pourrait en rendre l’utilisation plus accessible.

Utilisation

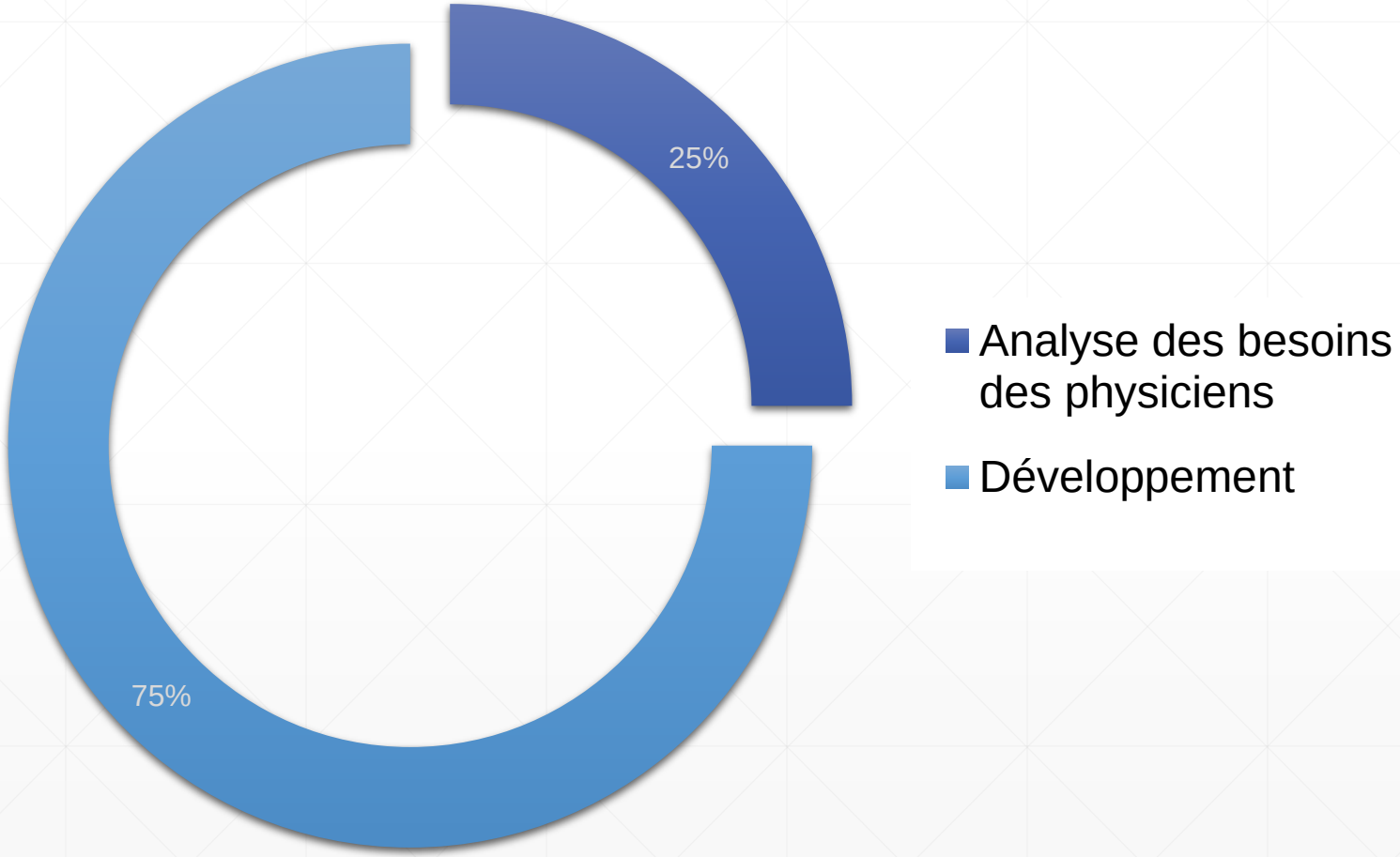
- Utilisateur: schématise son système, en une hiérarchie de modules capables de s’échanger des messages. (documentation fournie)
- Le logiciel simule alors les échanges entre les modules de manière synchrone afin de détecter toute erreur ou conflit.
- De plus une sortie des données générée est possible dans des fichiers de données formatées, facilement utilisables par un grapheur.


SATELLITE
Communication
Controlleur
Antenne
Ordinateur de bord
Module de pilotage
House keeping
Gestion énergie
Controlleur
Batterie

Choix d’implémentation

- Utilisation d’un langage à objets. Le C++ a été choisi, associé au système de documentation automatique Doxygen.
- Un noyau abstrait commun à tout type de simulation.
- La spécialisation ne se fait pas dynamiquement mais par héritage.
- Fonctionnement synchronisé de tous les modules qui communiquent entre eux via des message.
- Initialisation via des fichiers de configuration externes au format xml.
- Le projet est open-source et hébergé sur GitHub.

Une place importante laissée au dialogue

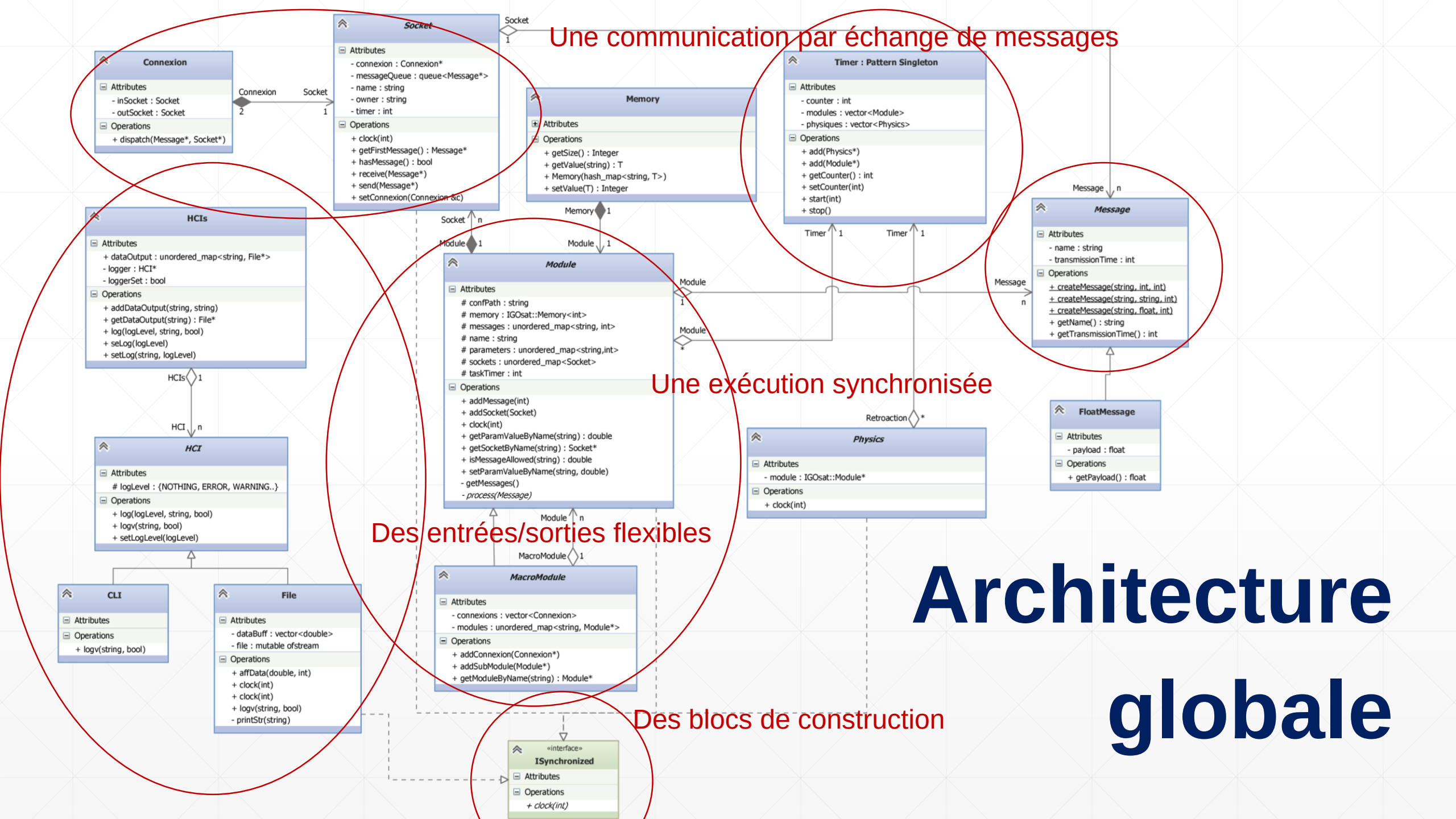


Répartition du travail

- Faite de manière informelle.
- Liste des tâches (incrémentales) établie en commun.
- Répartition des tâches selon les envies de chacun.

Compétences utilisées

- Notions avancées de programmation objet.
- Utilisation de Doxygen pour générer la documentation.
- Gestion d’un projet assez vaste faisant intervenir de nombreux interlocuteurs.
- Utilisation (somme toute assez basique) de Git.


Une communication par échange de messages
Une exécution synchronisée
Des entrées/sorties flexibles
Des blocs de construction
Architecture globale
Connexion
Attributes
- inSocket : Socket
- outSocket : Socket
Operations
+ dispatch(Message*, Socket*)
Socket
Attributes
- connexion : Connexion*
- messageQueue : queue<Message*>
- name : string
- owner : string
- timer : int
Operations
+ clock(int)
+ getFirstMessage() : Message*
+ hasMessage() : bool
+ receive(Message*)
+ send(Message*)
+ setConnexion(Connexion &c)
Memory
Attributes
Operations
+ getSize() : Integer
+ getValue(string) : T
+ Memory(hash_map<string, T>)
+ setValue(T) : Integer
Timer : Pattern Singleton
Attributes
- counter : int
- modules : vector<Module>
- physiques : vector<Physics>
Operations
+ add(Physics*)
+ add(Module*)
+ getCounter() : int
+ setCounter(int)
+ start(int)
+ stop()
Message
Attributes
- name : string
- transmissionTime : int
Operations
+ createMessage(string, int, int)
+ createMessage(string, string, int)
+ createMessage(string, float, int)
+ getName() : string
+ getTransmissionTime() : int
HCIs
Attributes
+ dataOutput : unordered_map<string, File*>
- logger : HCI*
- loggerSet : bool
Operations
+ addDataOutput(string, string)
+ getDataOutput(string) : File*
+ log(logLevel, string, bool)
+ seLog(logLevel)
+ setLog(string, logLevel)
Module
Attributes
confPath : string
memory : IGOsat::Memory<int>
messages : unordered_map<string, int>
name : string
parameters : unordered_map<string,int>
sockets : unordered_map<Socket>
taskTimer : int
Operations
+ addMessage(int)
+ addSocket(Socket)
+ clock(int)
+ getParamValueByName(string) : double
+ getSocketByName(string) : Socket*
+ isMessageAllowed(string) : double
+ setParamValueByName(string, double)
- getMessages()
- process(Message)
Physics
Attributes
- module : IGOsat::Module*
Operations
+ clock(int)
FloatMessage
Attributes
- payload : float
Operations
+ getPayload() : float
HCI
Attributes
logLevel : {NOTHING, ERROR, WARNING..}
Operations
+ log(logLevel, string, bool)
+ logv(string, bool)
+ setLogLevel(logLevel)
CLI
Attributes
Operations
+ logv(string, bool)
File
Attributes
- dataBuff : vector<double>
- file : mutable ofstream
Operations
+ affData(double, int)
+ clock(int)
+ clock(int)
+ logv(string, bool)
- printStr(string)
MacroModule
Attributes
- connexions : vector<Connexion>
- modules : unordered_map<string, Module*>
Operations
+ addConnexion(Connexion*)
+ addSubModule(Module*)
+ getModuleByName(string) : Module*
«interface»
ISynchronized
Attributes
Operations
+ clock(int)

```cpp
#pragma once
#include ...

typedef std::unordered_map<std::string, double> Params;
typedef std::unordered_map<std::string, Socket> Sockets;
typedef std::unordered_map<std::string, int> Messages;

class Module : public ISynchronized {
protected:
    std::string name;                       /*!< Le nom du module */
    std::string confPath;                   /*!< Le chemin vers le fichier de configuration */
    Memory<int> memory;                     /*!< La mémoire du module */
    Sockets sockets;                        /*!< Les connecteurs du module */
    Messages messagesAllowed;       /*!< Les messages compris par le module ET leurs temps d'exécution */
    Params parameters;                      /*!< Les paramètres d'état du modules */
    std::queue<std::shared_ptr<Message>> tasks;             /*!< La file d'attente des messages à traiter */
    int taskTimer;                          /*!< Le timer de la tâche courante */

public:
    Module(std::string = "DefaultName", Params = Params(), std::string cp = std::string());
    Module(std::string, Memory<int>, Params = Params());
    virtual ~Module(); virtual void clock(int);

    void addSocket(Socket);
    void addMessage(std::string, int);
    Socket* getSocketByName(std::string);
    double getParamValueByName(std::string);
    void setParamValueByName(std::string, double);
    bool isMessageAllowed(std::string);
    std::string getName() const;
    Socket* operator[](std::string);

private:
    void getMessages();
    virtual void process(std::shared_ptr<Message>) = 0;
};
```

La classe centrale

Module

```
void Module::clock(int time) {

    //On récupère un message par socket: getMessages();
    //On traite un peu: //Si fin attente tâche suivante:
    if (taskTimer == 0) {
        shared_ptr<Message> nextMsg = tasks.front();

        if (isMessageAllowed(nextMsg->getName())){
            //Process est virtuelle pure, dépend de chaque module !
            process(nextMsg);
            tasks.pop();

            //On met à jour le timer:
            if (tasks.size() > 0){
                taskTimer = messagesAllowed[nextMsg->getName()];
            } else { taskTimer = NOP; }
         } else{
            exit(EXIT_FAILURE);
        }
    } else {
         //Sinon, on dort:
        if(taskTimer != NOP){
            taskTimer--;
        }
    }
}
```

La méthode qui compte

void Module::clock(int)

Conclusion

- Aspect génie logiciel: importance de la documentation et interactions avec le client.
- Beaucoup de chemin parcouru, un projet intéressant qui sera peut-être utilisé par d’autres mini-satellites.
- Mais encore beaucoup à faire. Utilisation compliquée, et interactivité limitée.
- Le travail va continuer doucement, notamment le support technique. Peut-être dans le cadre d’un stage.

- Architecture statique vs dynamique.