**ESIL**

08

**Rapport PSR**

**WarThread**

**Florent Mas,Sylvain Paillé,Tony Quesnel,Aurélien Fiol,Michaël Lanöe, Jean-Marc Muzi**

Sommaire

[1 Introduction 3](#_Toc199003135)

[2 Présentation du jeu 3](#_Toc199003136)

[2.1 But 3](#_Toc199003137)

[2.2 Règles 4](#_Toc199003138)

[3 Entités du jeu 5](#_Toc199003139)

[3.1 Dieu 5](#_Toc199003140)

[3.2 Fidele 5](#_Toc199003141)

[3.3 Warrior 5](#_Toc199003142)

[3.4 Diagramme UML 5](#_Toc199003143)

[4 Création du réseau et déploiement 7](#_Toc199003144)

[4.1 Création du graphe 7](#_Toc199003145)

[4.2 Déploiement du graphe 7](#_Toc199003146)

[4.3 Visualisation du jeu 7](#_Toc199003147)

[5 Déroulement du Jeu 7](#_Toc199003148)

[5.1 Migration des warriors 7](#_Toc199003149)

[5.2 IA des warriors 7](#_Toc199003150)

[5.2.1 Récolte des informations 7](#_Toc199003151)

[5.2.2 Détermination d’une stratégie 7](#_Toc199003152)

[5.3 Gestion des batailles 8](#_Toc199003153)

[6 Gestion des pannes 8](#_Toc199003154)

[6.1 Simulation d’une panne 8](#_Toc199003155)

[6.2 Détection des pannes 8](#_Toc199003156)

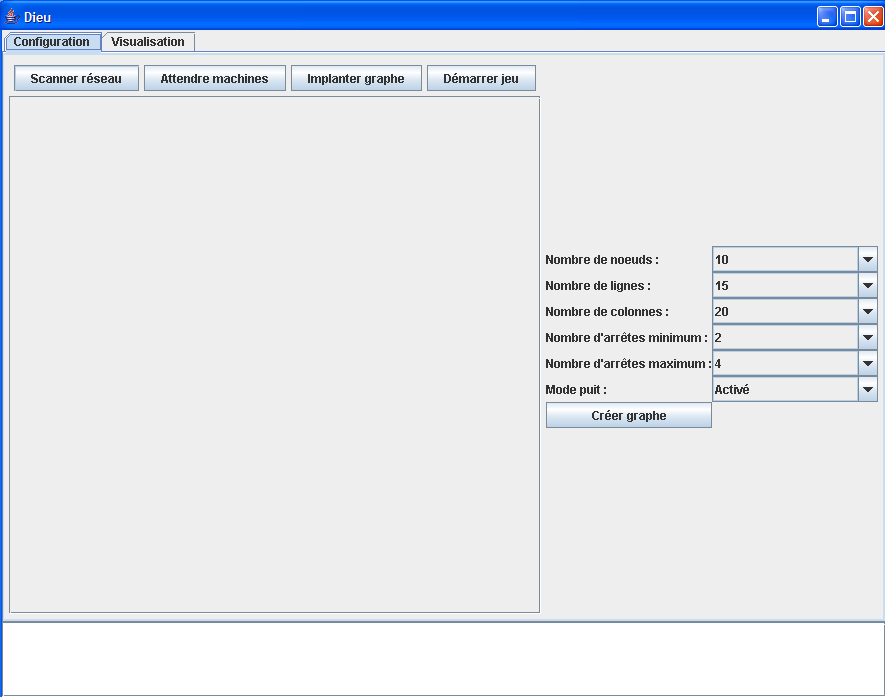
[6.3 Reconstruction d’un nœud 9](#_Toc199003157)

# Introduction

Notre objectif était de réaliser un système représentant une bataille entre thread. Le projet a été développé par une équipe de 6 personnes. Le projet aura pour but de mettre en œuvre les différents mécanismes vus en cours. Le projet reposera sur plusieurs parties notamment les entités du jeu, la création du graphe et son déploiement, les conditions de déroulement du jeu et une gestion des pannes. Ces parties seront présentées dans la suite de ce document.

# Présentation du jeu

## But



Le but du jeu est de remporter une bataille qui oppose plusieurs équipes constituées de thread. Pour cela, chaque équipe établit sa stratégie en décidant du comportement qu’auront ses threads face aux situations qu’ils peuvent rencontrer. Le Jeu se déroulera sur un graphe constitué d’un certain nombre de nœud. La fenêtre ci-dessus permet à l’un des joueurs de créer un graphe en lui donnant les options permettant son déploiement. Les joueurs auront la possibilité de visualiser le graphe et les déplacements des threads en sélectionnant l’onglet visualisation.

## Règles

Chaque équipe dispose d’un type de thread, ayant une stratégie qui lui est propre. Le jeu comporte deux catégories de nœud avec une base pour chaque équipe et des nœuds à capturer. La différence entre les bases et les nœuds est que les bases produisent à intervalles réguliers des threads pour appuyer l’armée des équipes.

Pour remporter une partie, les équipes ont plusieurs possibilités qui leur sont proposées :

-soit les deux équipes tentent de conquérir un maximum de nœud et le chronomètre de la partie se termine. Dans ce cas, l’équipe qui remporte la partie est celle qui a capturé le maximum de nœud.

-soit les deux équipes capturent les nœuds et au fil du jeu l’une des deux équipes se retrouve avec la totalité des nœuds de la carte en sa possession.

-soit l’une des deux équipes réussit à capturer la base de son adversaire.

# Entités du jeu

## Dieu

## Fidele

## Warrior

## Diagramme UML

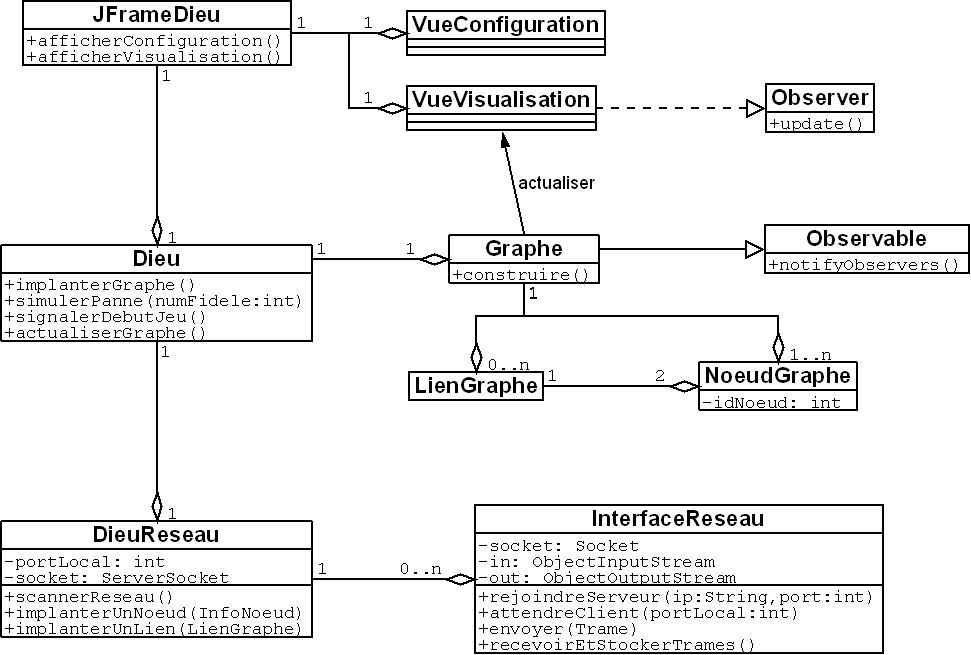


Figure : UML du programme DIEU

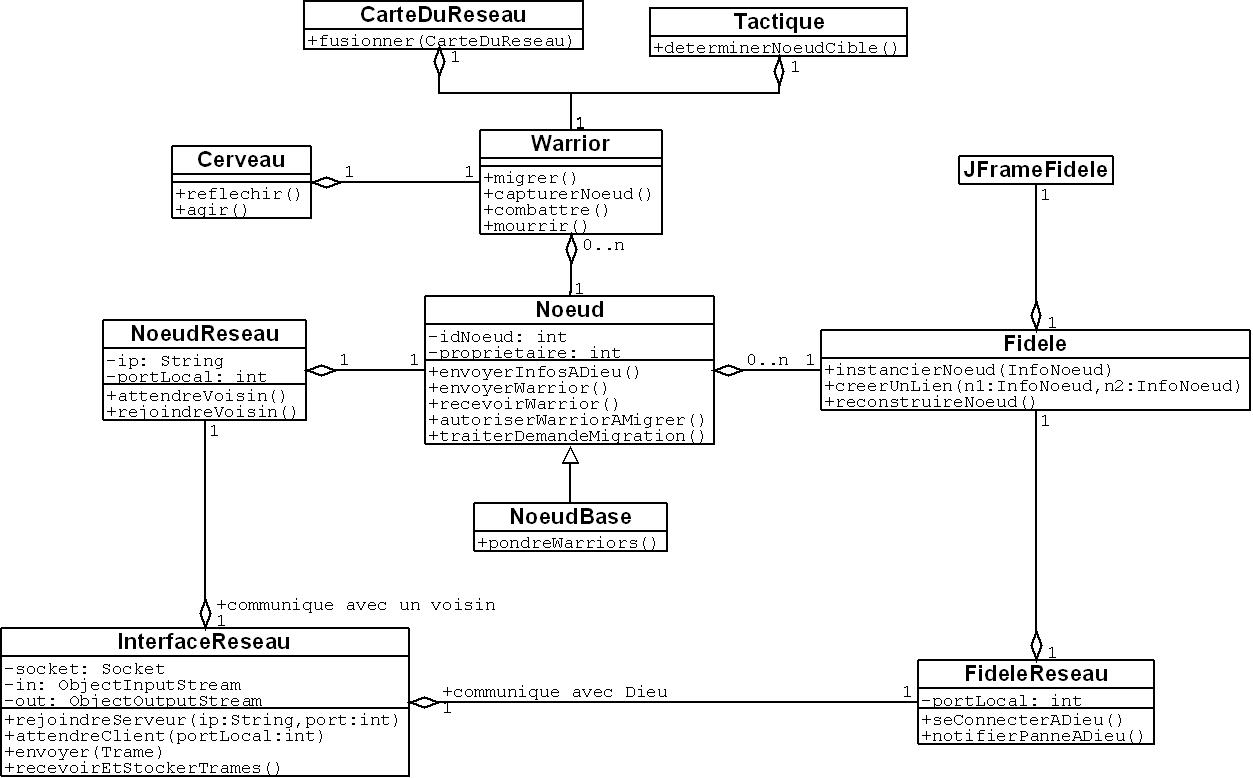


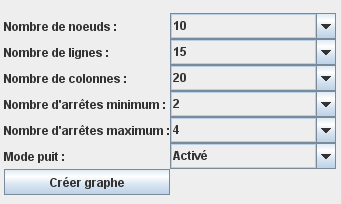
Figure : UML du programme Fidèle

# Création du réseau et déploiement

Le jeu se déroule sur un réseau virtuel représenté par un graphe connexe où à chaque sommet correspond un point de capture du jeu. Afin de réaliser ce réseau nous avons opté pour une solution élégante de génération aléatoire de carte à la demande. Suite à cette génération, les nœuds (sommets du graphe) sont dispatchés sur les différentes machines (fidèles) qui participent au support du jeu.

## Création du graphe

La création du graphe s’effectue du coté Dieu, par conséquent celle-ci s’effectue sur une et une seule machine. Les paramètres de création sont nombreux et permettent de générer à chaque lancement de l’algorithme un réseau virtuel particulier.



* **Nombre de nœuds (de 2 à 200)**

Le nombre de nœuds détermine la richesse du réseau virtuel crée. Il est préférable de choisir une valeur modérée afin de permettre à l’utilisateur du jeu de suivre son évolution sans s’y perdre.

* **Le nombre de lignes et le nombre de colonnes de la matrice**

L’algorithme dispose de manière aléatoire les nœuds sur une matrice N lignes et M colonnes

* **Le nombre d’arrêtes minimums et maximum**

Influe sur le nombre d’arrêtes (chemins pour les threads) entre un nœud et ses voisins.

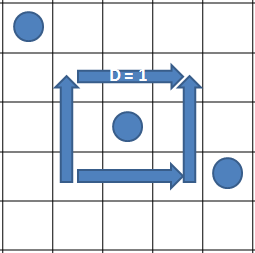
* **Le mode puît**

Le mode puît passe outre les limites du nombre d’arrêtes fixés et permet d’obtenir des topologies de graphe intéressantes.

En effet, dans ce mode, chaque nœud du graphe qui se connecte à un voisin doit aussi se connecter à tous les voisins de distances égales à celui-ci.

Concernant l’algorithme de création du graphe, nous avons implémenté différentes solutions qui donnaient des résultats plus au moins agréables visuellement, jusqu'à ce nous trouvions la solution actuelle dont l’algorithme est définie ci-dessous :

* On définie une matrices N lignes et M colonnes où chaque élément représente un emplacement possible pour un nœud du graphe. La taille de cette matrice est définie selon les paramètres décrits ci-dessus. Le nombre de nœud choisi doit être inférieur ou égal à la capacité (NxM) de la matrice.
* Nous définissons aussi un vecteur des cases libres qui contient les indices des emplacements libres de la matrice, ce vecteur est utilisé lors du choix aléatoire du choix nœud à disposer dans la matrice.
* Nous définissons une matrice d’adjacence qui nous permettra de vérifier une fois le graphe créée de la connexité de celui et dans le cas échéant d’entreprendre une procédure de réparation de la connexité.
* L’algorithme dispose de manière aléatoire les N nœuds sur la matrice du graphe en utilisant à chaque itération un élément libre de la matrice.
* Celui ci cherche ensuite des voisins auxquels se lier dans les limites du nombre d’arrêtes entré, cette recherche s’effectue par itération sur la distance en nombre d’emplacements qui séparent le nœud de ses voisins.



Ce choix est astucieux car il permet de connecter de préférence, les nœuds qui sont dans un premier lieu, les plus proches entre eux. En effet il fallait avoir à l’esprit qu’on nous devions obtenir en fin de compte un graphe lisible.

Néanmoins ce choix pose quelque peu problème car il ne garanti pas la connexité du graphe lorsque tous les nœuds sont disposés !

* C’est ici que rentre en jeu la détection de la connexité grâce à l’application de l’algorithme de Floyd-Warshall sur la matrice d’adjacence afin de calculer sa fermeture et de déterminer les ensemble de graphes non connexes. Par la suite il suffit de relier les deux nœuds les plus proches des deux graphes par une arrête.
* Pour finir, nous choisissons deux nœuds qui serviront de bases respectives aux joueurs, les deux bases choisies sont les deux nœuds du graphe dont la distance respective est la plus élevée.

Voici ci-dessous le résultat visuel de la création d’un graphe avec 100 bases, les bases des joueurs ( vertes et rouges ) sont aux antipodes du graphe.

INSERER GRAPHE AVEC 100 NŒUDS + BASES

## Déploiement du graphe

A l’issue de l’étape de création nous disposons de deux éléments importants :

* La liste des nœuds (leur emplacement physique sur le réseau)
* La matrice d’adjacence (qui marque les connexions entre les nœuds)

Ces deux éléments nous permettent de dispatcher des parties de graphes (ensemble de nœuds quelconques) à une liste de machines qui serviront donc chacune de support au bon fonctionnement du jeu.

Les machines qui supporteront les nœuds du graphe sont des machines clientes sur lesquelles le programme fidèle a été installé. Leur participation au jeu se fait de deux manières différentes :

* Par demande explicite de la demande de participation au jeu. Cette procédure requiert une configuration manuelle de chaque fidèle (avec explicitation de l’adresse IP de Dieu) .
* Par attente passive de la demande de participation. Ce mode est celui qui est principalement utilisé, Dieu scanne le réseau local à la recherche de machines clientes susceptibles d’héberger les nœuds du graphe.

Voici d’un point de vue fidèle, les deux approches citées ci-dessus.



Du coté Dieu, les deux boutons correspondants sont : 

Voici une capture d’écran de quelques machines clientes du jeu :

INSERER SCREENSHOT LISTE MACHINES

Une fois la détection des machines clientes effectuée, il est alors nécessaire de déléguer à chacune d’entres elles un certain nombre de nœuds.

Cette délégation se fait de manière aléatoire, Dieu envoie alors a chaque machine les informations nécessaire à la création des nœuds sur les machines distantes. Ce processus s’effectue lors d’un clic sur le bouton

La communication entre fidèles et plus généralement dans l’ensemble du projet s’effectuent exclusivement par socket, des trames de types différentes permettent de recevoir et d’envoyer des informations de types différents.

En ce qui concerne le déploiement des nœuds nous envoyons deux types de trames :

* Trame *NOUVEAU\_NOEUD*

Cette trame contient les informations sur le nœud à créer

* Trame *NOUVEAU\_LIEN*

Cette trame contient les informations sur une arrête unique reliant deux nœuds

Une fois l’implantation terminée, le jeu peut commencer suite à un clic sur le bouton 

## Visualisation du jeu

Dans l’énoncé original du projet, une proposition de visualisation était d’afficher des fenêtres de couleurs différentes lors de la capture d’un nœud. Nous avons pensé que cette approche ne permettait pas d’avoir une vue globale de l’avancement du jeu et nous avons préféré centraliser cette visualisation au niveau de Dieu.

Nous nous servons donc de Dieu comme observateur du jeu. Pour rappel, Dieu connaît déjà la topologie du réseau car c’est lui qui a dispatché les nœuds à ses fidèles.

Le graphe crée est affiché à l’aide de JGraph et de la surcharge des méthodes d’affichages des cellules afin de représenter les nœuds. Les threads sont représentés à l’aide de bâtonnets verts ou rouge selon leur appartenance.

La communication entre Dieu et les fidèles s’effectuent à l’aide de socket et de FIFO. En effet, chaque changement d’état d’un nœud implique l’envoie d’une trame d’information sur le nœud modifié à Dieu. Les trames peuvent être nombreuses et Dieu les traite les unes aux suites des autres dans leur ordre d’arrivée.

Dieu reçoit 2 types de trames liées à la visualisation du jeu :

* Trame *INFO\_NOEUD*

L’état du nœud à été modifié (connecté, capturé, nouveau thread, suppression de thread)

* Trame *INFO\_THREAD*

Un thread a décidé de migrer, on représente donc graphiquement le déplacement.

# Déroulement du Jeu

## Migration des warriors

## IA des warriors

Le déroulement du jeu nécessite un comportement intelligent des warriors (agents mobiles).Nous avons donc implémenté une IA respectant le schéma classique de réflexion d’une entité intelligente :

Cette IA va donc récolter des informations sur la carte présente et agir en conséquence.

### Récolte des informations

Afin de pouvoir créer des stratégies sophistiqués, il était intéressant d’éviter toute approche déterministe où chaque thread crée aurait le même plan d’action que tous les autres. Pour cela nous avons introduit deux notions dans le jeu :

* Les threads peuvent communiquer entre eux lorsqu’ils se rencontrent sur des nœuds du réseau
* Ils collectent des informations sur les nœuds visités

Ces informations sont très importantes et permettent d’établir des matrices de risques et des tactiques d’attaque, de défense ou d’exploration d’une partie du graphe si nécessaire.

Chaque thread se déplace de nœud en nœud avec un bagage important d’informations :

* Une matrice des distances des nœuds parcourus
* Une liste de dernières informations collectées
  + Nombre de threads alliés
  + Nombre de threads ennemis
  + Propriétaire du nœud
  + Date de la dernière collecte

Lors de chaque rencontre avec des threads alliés un processus d’échange s’effectue :

* Les threads fusionnent leurs informations respectives sur les nœuds, les informations les plus récentes sont alors partagées par tous.
* Ils échangent aussi leur matrice des distances, deux threads peuvent parcourir respectivement une partie de graphe puis s’informer des nœuds rencontrés.
* Nous appliquons de nouveau Floyd-Warshall sur chaque matrice des distances afin cette fois ci de calculer les distances minimales d’un nœud à l’autre.

Nous déduisons de ces informations un tableau de risque indiquant le risque entrepris pour aller du nœud courant à un autre nœud de la carte (pas forcement voisin !)

Voici ci-dessous un schéma représentation l’échange de deux cartes, le calcul des distances minimales n’est pas montré.

Lors de la sortie de ce processus, un thread a possession de tout le nécessaire afin d’établir une stratégie en connaissances des risques (lenteur des déplacements + nombre de threads ennemis sur le nœud cible)

Il est à noter qu’en dehors de l’usage tactique de ces informations, nous avons créée un système de calcul distribué des distances minimales d’un graphe quelconque. En effet il faut rappeler que les calculs s’effectuent sur des fidèles différents donc sur des machines différentes. Notre travail s’inclut donc dans un cadre plus général que celui du jeu en lui-même.

### Détermination d’une stratégie

Après la récolte des différentes informations sur la carte. L’IA va choisir un type de stratégie. Il en existe 4 :

* Stratégie indéterminée
* Stratégie Exploratrice
* Stratégie Offensive
* Stratégie Défensive

Le choix va se faire sur différent facteurs : le risque des nœuds aux alentours, le coefficient d’agressivité de l’IA du joueur et le nombre de nœud connus par le warrior.

1. La stratégie Indéterminée : elle permet au warrior d’évoluer sur le réseau de manière aléatoire au début du jeu ou dans le cas ou aucune autre stratégie ne serait applicable.
2. La stratégie Exploratrice : Cette stratégie choisit toujours d’explorer les nœuds inconnus : c’est-à-dire que parmi les voisins du nœud sur lequel se trouve le warrior, elle va choisir un nœud qui n’est pas présent dans la carte du warrior.
3. La stratégie offensive : Elle permet au warrior d’attaquer des nœuds qui possèdent un coefficient « risque » plus ou moins élevé : c'est-à-dire un nœud qui a été capturé par l’ennemi ou accueillant un ou plusieurs warrior ennemis.  
   Ce choix se fait parmi toute la carte en possession du warrior .La stratégie calcule donc **le chemin le plus court** pour accéder au nœud grâce à **l’algorithme de Dijkstra.**Si plusieurs warriors sont nécessaires pour la capture du nœud, le thread attendra un certains laps de temps que d’autres warriors de son camp viennent l’aider.
4. La stratégie offensive : Cette stratégie se met en place lorsque le risque de la majorité des nœuds connus est beaucoup trop élevé. Le warrior va alors défendre sa base puis les nœuds aux alentours.  
   De la même manière que la stratégie offensive les chemins sont calculés avec dijkstra.

Chaque stratégie définit donc une ou plusieurs actions qui sont :

* Migrer directement vers le nœud cible
* Migrer vers le nœud cible en passant par un chemin donné
* Attaquer le nœud
* Attendre l’arrivé de renfort

## Gestion des batailles

### Surveillance de nœuds

Chacun des nœuds composant le terrain du jeu, à sa création lance une fonction de surveillance, chargée d’analyser en permanence (à de très faibles intervalles) l’activité présente sur son nœud afin de juger si un combat est nécessaire. Cette surveillance s’effectue dans un thread externe, afin de ne pas interférer ou bloquer l’action du jeu courante.

Cette tache de surveillance consiste à parcourir l’ensemble des threads Warriors présents actuellement sur le nœud, d’observer leur joueur propriétaire et de contrôler leur état : veulent-ils capturer le nœud ou sont-ils seulement de passage. En fonction de ces informations, la fonction prend une décision :

* Se trouvent présents sur le nœud au moins deux Warriors de propriétaire différents, voulant capturer le nœud. On appelle alors la fonction de résolution de combat qui prendra le relais.
* Des Warriors veulent capturer le nœud, et ils sont tous du même camp : si ils respectent toutes les conditions requises pour capturer le nœud, alors celui-ci change de camp.
* Dans toutes autres situations, la fonction de surveillance n’agit pas et continue de surveiller.

### Combats de threads

Lorsque des Warriors d’au moins deux camps différents désirant prendre contrôle du nœud se trouvent côte à côte, un combat s’engage pour déterminer quel sera le camp gagnant, et quels seront les perdants.

Afin de décider l’issue de la bataille, chacun des Warriors de chaque camp lance un dé. Le nombre aléatoire de 1 à 100 sortants de ce lancer représente la force que les Warriors mettent dans la bataille. Après avoir fait la somme des lancers des Warriors pour chaque camp, on rajoute un bonus pour le camp possédant actuellement le nœud (dans le cas où le nœud n’est pas neutre et que le camp propriétaire fasse partie du combat en cours) : on ajoute un lancer de dés d’autant de fois que du montant de l’attribut ‘coefficient’ du nœud sur lequel l’action se déroule. Cet attribut sert à diverses choses, notamment à savoir la quantité de Warriors nécessaires pour capturer le nœud, ou comme ici de bonus au camp qui a déjà capturé le dit nœud.

Une fois que le compte est fait, il est facile de déterminer le vainqueur du combat : il s’agit du camp totalisant le plus grand score – s’il y a égalité, ce sera celui de plus petit identificateur.

Le camp gagnant capture le nœud. On parcourt une dernière fois la liste des Warriors, et tout ceux appartenant au camp ayant totalisé le plus petit résultat doivent mourir. Il n’est pas nécessaire de continuer à se battre jusqu’à ce qu’il ne reste plus qu’un camp : si cela est nécessaire la fonction de surveillance du nœud rappellera la fonction de résolution de combat.

Nous avons choisi de résoudre les batailles de cette manière, car elle offre un grand intérêt stratégique ainsi qu’un bon compromis : plus un camp dispose de Warriors de son côté et plus elle a de chances de remporter la bataille ; mais rien n’est jamais joué à l’avance et la chance et le hasard interviennent tout de même pour rendre les bataille toujours intéressantes et inattendues. De plus, grâce au bonus apporté au camp propriétaire, plus un nœud est difficile à capturer, et plus le camp gagnant aura de facilité à le défendre et de chance de conserver sa propriété.

### Capture du nœud

Lorsqu’un combat se termine ou qu’un camp seul sur un nœud décide de le conquérir, il faut alors décider si le nœud est capturé ou non.

L’attribut ‘coefficient’ du nœud sert principalement à cela : il représente le nombre nécessaires pour contrôler le nœud. Si la fonction de surveillance ne détecte que des Warriors d’un même camp, et que ces Warriors veulent capturer le nœud courant, alors elle compare leur nombre au ‘coefficient’ du nœud ; si ils sont supérieur ou également nombreux, alors le nœud prend comme propriétaire celui de ces Warriors, et cela qu’il soit neutre ou appartienne déjà à un camp adverse.

Dans le cas d’un combat de Warriors, c’est assez différent : la fonction de résolution de combat, à l’issue de la bataille, donne le nœud au camp victorieux, et cela sans prendre en compte le nombre de Warriors ou le ‘coefficient ‘ du nœud. Le fait de s’être battu à la fois les un contre les autres mais aussi parfois contre le terrain lui-même affranchit les joueurs d’une telle règle. Cela peut-être très intéressant d’un point de vue stratégique…

# Gestion des pannes

## Simulation d’une panne

## Détection des pannes

Pour détecter une panne, nous avons mis en place un système pour migrer un thread d’un nœud A vers un nœud B. La migration d’un thread nécessite l’envoie d’une demande d’autorisation de migration. Si la demande a bien été reçue, le nœud B va envoyer une réponse au nœud A. S’il n’y a aucune réponse, alors nous considérons que le nœud B est en panne.

Quand la panne est détectée, le nœud qui l’a découvert envoie une trame à Dieu pour le prévenir de l’incident. Cette trame contient l’identifiant du nœud en panne. Une fois que Dieu est informé de la panne, il envoie une trame à chaque voisin du nœud en panne ainsi qu’au fidèle sur lequel était le nœud en panne pour leur spécifier que le nœud d’identifiant k est en panne. A ce moment là, il y a une partie de l’IA qui s’occupe de traiter comment les voisins du nœud en panne vont traiter les threads notamment pour les migrations. Le but étant d’éviter de faire migrer un thread vers un nœud en panne et de surcharger le réseau en prévenant Dieu à plusieurs reprises que qu’un nœud k est en panne.

## Reconstruction d’un nœud

La reconstruction du nœud est faite par le fidèle sur lequel se trouvait le nœud avant de tomber en panne. Pour cela Dieu va lui envoyer une trame avec l’identifiant du nœud et la valeur du propriétaire avant la panne. Le fidèle reconstruira un nœud identique à celui en panne c'est-à-dire en respectant les liens qui le reliaient à ses voisins.