

DotAI - Projet de semestre

Thomas Ibanez

5 mars 2018

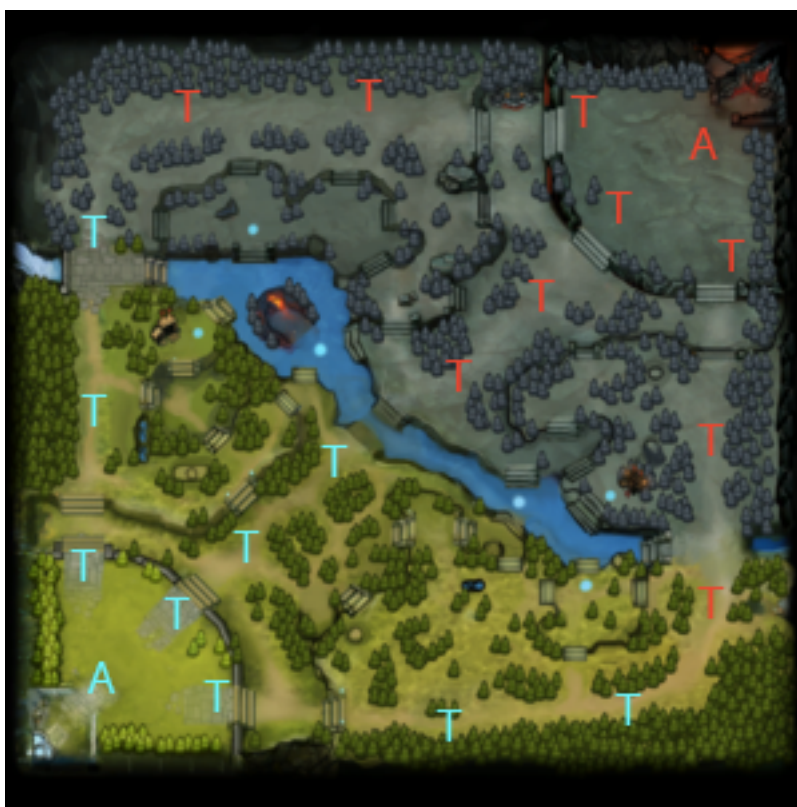
Table des matières

1 Introduction

Le but de ce projet de semestre est de créer un ensemble de logiciels permettant de présenter les données contenues sur `opendota.com` pour le développement d'un logiciel d'apprentissage automatique sur le jeu DotA 2.

1.1 DotA 2

DotA 2 est jeu-vidéo multijoueur de type bataille en arène (MOBA) où 2 équipes s'affrontent. Chaque équipe est composée de 5 joueurs chacun contrôlant un héros choisi parmi les héros disponibles dans le jeu (il ne s'agit donc pas d'un avatar personnalisable comme dans le cas des MMORPG). A l'heure actuelle le jeu propose 115 héros, chacun disposant de 5 attaques différentes. La partie se déroule sur une carte symétrique constitué de 3 "lanes" sur lesquelles sont disposées des tours dont voici une vue aérienne :



Chaque équipe commence dans un coin de la carte (en bas à gauche et en haut à droite), le but pour chaque héros est de contrôler une partie de la carte et détruire les tours ennemies (T) afin d'arriver jusqu'à la base adverse (A) et de la détruire.

En plus des héros chaque base va périodiquement créer des "creeps", sortes de petit monstres qui vont se déplacer sur une lane.



2 Récupération des informations d'OpenDota

Opendota collecte une quantité énorme de parties de DotA 2 jouées par des joueurs du monde entier. Le site présente une API permettant la récupération de matchs selon certains critères. Une documentation sur le genre de requêtes faisable est disponible sur <https://docs.opendota.com/>. Dans le cadre de se projet, nous allons utiliser un filtre permettant de sélectionner un héros dont on voudrait qu'il soit présent dans le match.

Dans un premier temps M. Malaspinas et moi-même pensions que les données étaient directement exploitable depuis opendota, malheureusement c'est plus compliqué. En effet OpenDota stocke uniquement des données statistiques sur la partie jouée mais pas les actions et positions des entités à chaque instant. Heureusement le résultat contient un lien vers un fichier *.dem* qui lui contient toutes ces informations.

3 Analyse des fichiers *.dem*

Les fichiers *.dem* sont des fichiers contenant toutes les informations dont on pourrait vouloir sur une partie allant de entités présentes sur la carte jusqu'au commentaires audio si disponibles. Un fichier *.dem* pour une partie standard (~30 Minutes) pèse ~50Mo.

3.1 Format

Le format global de ces fichiers est assez simple. DotA utilise protobuf, une technologie de sérialisation développée par google, pour encoder ces fichiers. Le protocole est maintenu à jour par l'équipe SteamRE sur <https://github.com/SteamRE/SteamKit/tree/master/Resources/Protobufs/dota>

Chaque fichier commence par une en-tête, soit "PBDEMS2" pour les replay datant d'après la mise à jour de DotA sur le moteur de jeu Source 2. Soit "PBUFDEM" pour les replay plus anciens. Dans le cadre de ce projet nous nous concentrons uniquement sur les replay d'après Source 2 car ce sont ceux qui sont émis en ce moment.

Une fois ce header lu le fichier contient 8-bytes dont l'utilité n'est pas connue.

3.1.1 Format global

La suite du fichier est une série d'entrées de ce type :

Nom	Type
ID	VarInt
Tick	VarInt
Taille	VarInt
Données	Bytes

Note : Les varints sont des entiers encodés sur une longueur variable de bits. Cet encodage est détaillé sur <https://developers.google.com/protocol-buffers/docs/encoding#varints>

L'ID va indiquer le type du message parmi la liste définie dans le protocole :

Messages
<pre>enum EDemoCommands { DEM_Error = -1; DEM_Stop = 0; DEM_FileHeader = 1; DEM_FileInfo = 2; DEM_SyncTick = 3; DEM_SendTables = 4; DEM_ClassInfo = 5; DEM_StringTables = 6; DEM_Packet = 7; DEM_SignonPacket = 8; DEM_ConsoleCmd = 9; DEM_CustomData = 10; DEM_CustomDataCallbacks = 11; DEM_UserCmd = 12; DEM_FullPacket = 13; DEM_SaveGame = 14; DEM_SpawnGroups = 15; DEM_Max = 16; }</pre>

```

    DEM_IsCompressed = 64 ;
}

```

La seule exception est *DEM_IsCompressed* qui n'est pas un type de message mais qui définit un bit (6ème bit) qui s'il est égal à 1, alors le message doit être décompressé via la librairie snappy avant d'être interprété.

Le Tick va définir le moment dans la partie où le message arrive.

Le champ Taille indique la taille en bytes des données de ce message.

Une fois ces informations connues nous pouvons lancer le décodage via protobuf afin d'obtenir le message, du moins c'est le cas pour tous les ID sauf DEM_SignonPacket, DEM_Packet, DEM_FullPacket et DEM_SaveGame. Pour ces messages-ci il va falloir lire les données intégrées.

3.1.2 Données intégrées

Les données intégrées sont basiquement un message encodé dans un autre message. Dans notre cas un DEM_Packet et un DEM_FullPacket sont définis comme :

```

                                Packets
message CDemoPacket {
    optional int32 sequence_in = 1 ;
    optional int32 sequence_out_ack = 2 ;
    optional bytes data = 3 ;
}

message CDemoFullPacket {
    optional .CDemoStringTables string_table = 1 ;
    optional .CDemoPacket packet = 2 ;
}

```

Les DEM_SignonPacket n'ont pas de définition distincte, leur structure est la même qu'un DEM_Packet. Les DEM_FullPacket sont en fait composés d'une *StringTable* et d'un DEM_Packet qu'il faudra interpréter comme les autres DEM_Packet.

Comme on peut le constater un packet n'est en vérité que composé de numéros de séquences (qui ne sont pas intéressants dans notre cas) et d'une suite de bytes qu'il nous faut interpréter. L'interprétation de ce champ est la suivante : Il s'agit d'une série de sous-messages dont encodés d'une manière ressemblant au format global du fichier :

Nom	Type
ID	Bitvar
Taille	VarInt
Données	Bytes

Ce qu'il faut remarquer c'est que l'ID est encodé en tant que "BitVar" et non pas en tant que VarInt. Ce format est un petit peu particulier et implique un désalignement :

Byte No:		0		1		2		3		4	//				
Bit No:		0 1 2 3		4 5 6 7		0 1 2 3		4 5 6 7		0 1 2 3		4 5 6 7		0 1	//
		-----		-----		-----		-----		-----		-----		---	//
Valeur:		X Y 0 0		0 0 1 1		1 1 2 2		2 2 3 3		3 3 3 3		3 3 3 3		3 3 3 3	//

Les bits "X", "Y" et "0" doivent obligatoirement être lus. Ensuite :

- Si Y vaut 1, il faut lire 4 bits supplémentaires (les bits "1")
- Si X vaut 1, il faut lire 8 bits supplémentaires (les bits "1" et "2")
- Si X et Y valent 1, il faut lire 28 bits supplémentaires (les bits "1", "2" et "3")

Note : Comme on peut le constater, la lecture de cette variable ne s'arrête non pas à la fin d'un byte mais bel et bien au milieu. Nous ne sommes donc plus alignés sur une base de 8 bits ce qui m'a obligé à créer un flux de lecture bit par bit au lieu d'utiliser un des flux disponibles avec java qui sont tous byte par byte.

La valeur est ensuite construite de la manière suivante : Les 4 bits "0" sont les bits de poids faible, les autres bits restent dans l'ordre dans lequel ils sont écrits (les "1" en poids fort).

Le champs taille définit la taille des données de ce sous-packet. Les données sont à nouveau une structure encodée via protobuf, cette structure peut être de type NET_Message ou SVC_Message (également d'autres types possible mais ils ne nous intéressent pas).

Voici les enums SVC_Messages et NET_Messages :

Messages

```

enum SVC_Messages {
    svc_ServerInfo = 40 ;
    svc_FlattenedSerializer = 41 ;
    svc_ClassInfo = 42 ;
    svc_SetPause = 43 ;
    svc_CreateStringTable = 44 ;
    svc_UpdateStringTable = 45 ;
    svc_VoiceInit = 46 ;
    svc_VoiceData = 47 ;
    svc_Print = 48 ;
    svc_Sounds = 49 ;
    svc_SetView = 50 ;
    svc_ClearAllStringTables = 51 ;
    svc_CmdKeyValues = 52 ;
    svc_BSPDecal = 53 ;
    svc_SplitScreen = 54 ;
    svc_PacketEntities = 55 ;
    svc_Prefetch = 56 ;
    svc_Menu = 57 ;
    svc_GetCvarValue = 58 ;
    svc_StopSound = 59 ;
    svc_PeerList = 60 ;

```

```

    svc_PacketReliable = 61 ;
    svc_HLTVStatus = 62 ;
    svc_FullFrameSplit = 70 ;
}

enum NET_Messages {
    net_NOP = 0 ;
    net_Disconnect = 1 ;
    net_SplitScreenUser = 3 ;
    net_Tick = 4 ;
    net_StringCmd = 5 ;
    net_SetConVar = 6 ;
    net_SignonState = 7 ;
    net_SpawnGroup_Load = 8 ;
    net_SpawnGroup_ManifestUpdate = 9 ;
    net_SpawnGroup_SetCreationTick = 11 ;
    net_SpawnGroup_Unload = 12 ;
    net_SpawnGroup_LoadCompleted = 13 ;
}

```

Dans ce projet, étant donné que nous ne cherchons pas à créer un interpréteur complet, ce sont les messages `svc_CreateStringTable`, `svc_UpdateStringTable`, `svc_PacketEntities` et `svc_ServerInfo` qui vont nous intéresser par la suite.

3.2 Données

Revenons au début du fichier, directement après le magic et les 8-bytes à sauter nous allons trouver un message de type `DEM_FileHeader`. Ce message contient des méta-données sur le fichier, par exemple :

```

demo_file_stamp: "PBDEMS2\000"
network_protocol: 44
server_name: "Valve Dota 2 Europe Server (srcds122.133.57)"
client_name: "SourceTV Demo"
map_name: "start"
game_directory: "/opt/srcds/dota/dota_v2432/dota"
fullpackets_version: 2
allow_clientside_entities: true
allow_clientside_particles: true
addons: ""

```

Ces informations ne nous sont cependant pas utiles.

Nous trouvons ensuite une série de `DEM_SignonPacket`, l'un d'entre eux contient un sous-message de type `svc_ServerInfo` dont voici la structure :

```

message CSVCMsg_ServerInfo {
    optional int32 protocol = 1 ;
    optional int32 server_count = 2 ;
}

```



```

optional bool is_dedicated = 3;
optional bool is_hltv = 4;
optional bool is_replay = 5;
optional int32 c_os = 6;
optional fixed32 map_crc = 7;
optional fixed32 client_crc = 8;
optional fixed32 string_table_crc = 9;
optional int32 max_clients = 10;
optional int32 max_classes = 11;
optional int32 player_slot = 12;
optional float tick_interval = 13;
optional string game_dir = 14;
optional string map_name = 15;
optional string sky_name = 16;
optional string host_name = 17;
optional string addon_name = 18;
optional .CSVCMsg_GameSessionConfiguration game_session_config = 19;
optional bytes game_session_manifest = 20;
}

```

Le champ `game_dir` va nous permettre de déterminer la version de DotA 2 utilisée, ce qui nous sera utile par la suite. Le contenu du champ ressemble à ça `"/opt/srcds/dota/dota_v2432/dota"` (à noter que ce champ est également présent dans le message `DEM_FileHeader`. La partie qui nous intéresse est `"v2432"` qui nous indique que le replay a été créé avec la version 2432 de DotA 2.

Nous trouvons ensuite des `DEM_Packets` qui contiennent des sous-messages de type `svc_CreateStringTable` qui vont servir, comme leur nom l'indique à créer des "String Tables".

3.2.1 String Tables

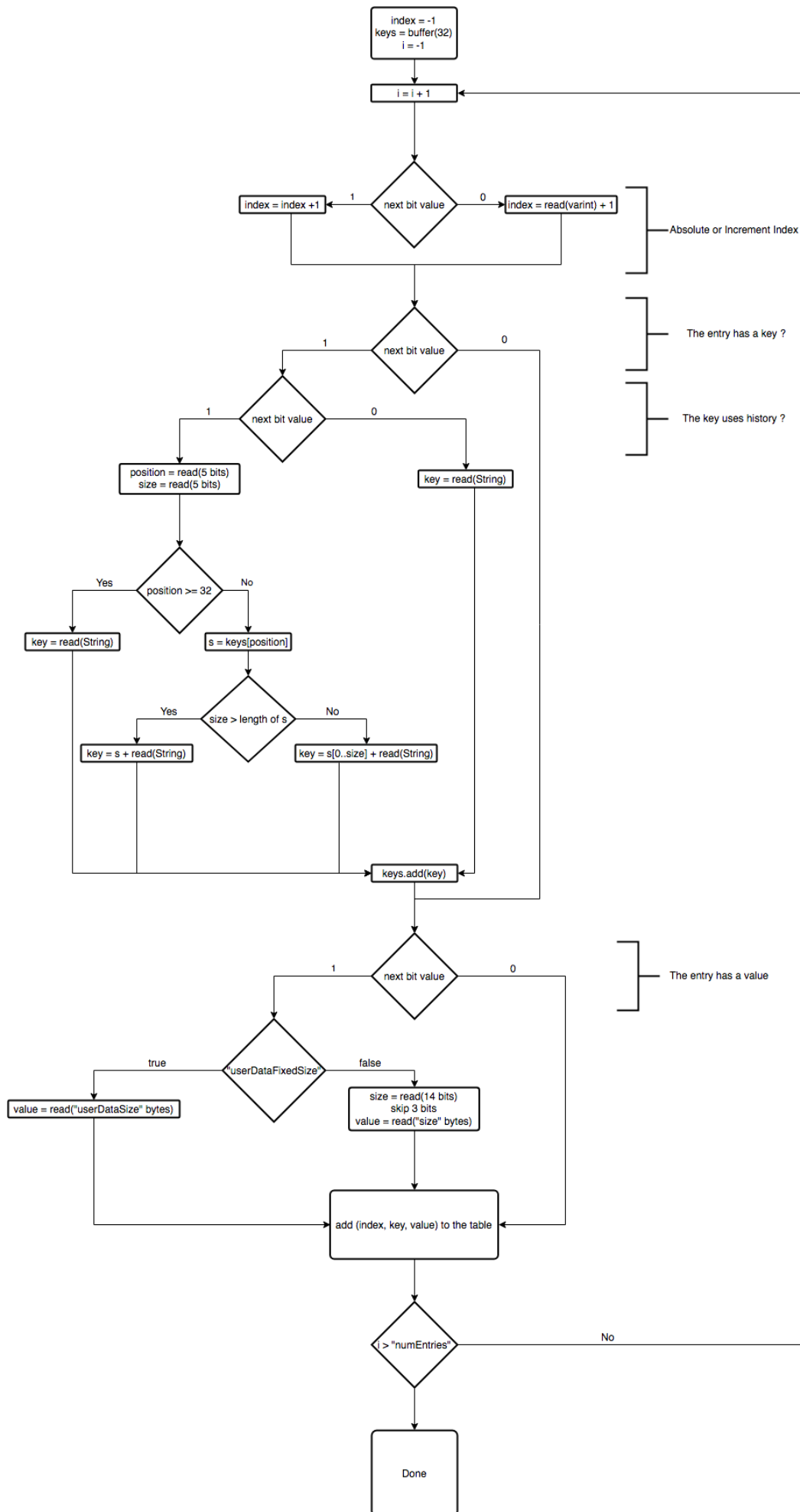
Une String Table est une table nommée, composée d'une série d'entrées ayant un index (entier), une clef (chaîne de caractères) et une valeur (chaîne de bytes) bien que la clef comme la valeur puisse ne pas être définie (null). Voici pour illustrer une petite partie de la String Table "EntityNames" (la table entière contient 251 entrées)

Index	Clef	Valeur
24	dark_troll_warlord_ensnare	null
25	dark_troll_warlord_raise_dead	null
26	polar_furboleg_ursa_warrior_thunder_clap	null

Cependant, pour ce projet la seule table qui va nous intéresser est la table "instancebaseline" qui contient les identifiants de toutes les entités qui ont des propriétés à lire avant leur création.

La totalité du contenu de la String Table est englobé dans le champ *StringData*, cependant il est possible que ce champ soit compressé avec la librairie Snappy pour les replay plus récents ou avec LZSS pour les plus anciens. Pour savoir quelle méthode est utilisée on peut vérifier si les 4 premiers bytes composent la chaîne de caractères "LZSS".

Une fois décompressé si besoin nous pouvons commencer la lecture, il nous faut dans un premier temps déclarer un buffer circulaire de 32 éléments qui va constituer un historique des clefs lues.



Une fois toutes ces String tables lues nous allons recevoir un packet contenant les "Send Tables" qu'il va falloir lire également.

3.2.2 Send Tables

La définition d'une SendTable est très simple :

```
message CDemoSendTables {  
    optional bytes data = 1;  
}
```

Comme on peut le constater il n'y a qu'un champ *data* qui est une suite de bytes, il va donc falloir le décoder.

Heureusement le décodage du champ *data* est simple, il faut d'abord lire un varint qui va nous indiquer une taille, puis lire cette taille (en bytes) afin d'obtenir une structure de type `svc_FlattenedSerializer`

```
message CSVCMsg_FlattenedSerializer {  
    repeated .ProtoFlattenedSerializer_t serializers = 1;  
    repeated string symbols = 2;  
    repeated .ProtoFlattenedSerializerField_t fields = 3;  
}  
  
message ProtoFlattenedSerializerField_t {  
    optional int32 var_type_sym = 1;  
    optional int32 var_name_sym = 2;  
    optional int32 bit_count = 3;  
    optional float low_value = 4;  
    optional float high_value = 5;  
    optional int32 encode_flags = 6;  
    optional int32 field_serializer_name_sym = 7;  
    optional int32 field_serializer_version = 8;  
    optional int32 send_node_sym = 9;  
    optional int32 var_encoder_sym = 10;  
}  
  
message ProtoFlattenedSerializer_t {  
    optional int32 serializer_name_sym = 1;  
    optional int32 serializer_version = 2;  
    repeated int32 fields_index = 3;  
}
```

Les flattened serializers vont nous permettre de connaître toutes les structures qui vont être lues par la suite, les champs qui les composent et leurs types respectifs. En revanche ils n'expliquent en rien la façon dont il faudra lire les différents types.

Voici un exemple de structure (`CBodyComponentBaseModelEntity`) et ses champs avec leurs types.

```

Serializer: CBodyComponentBaseModelEntity
    m_cellX uint16
    m_cellY uint16
    m_cellZ uint16
    m_vecX CNetworkedQuantizedFloat
    m_vecY CNetworkedQuantizedFloat
    m_vecZ CNetworkedQuantizedFloat
    m_hParent CGameSceneNodeHandle
    m_angRotation QAngle
    m_flScale float32
    m_nOutsideWorld uint16
    m_hModel CStrongHandle< InfoForResourceTypeCModel >
    m_MeshGroupMask uint64
    m_nDebugIndex int32
    m_nIdealMotionType int8
    m_ProceduralTargetContexts CUtlVector< CNetworkedIKProceduralTargetContext >
    m_name CUtlStringToken
    m_hierarchyAttachName CUtlStringToken
    m_bIsRenderingEnabled bool
    m_bIsAnimationEnabled bool
    m_materialGroup CUtlStringToken
    m_nHitboxSet uint8

```

A noter que certains types ne sont pas atomiques mais référencent une autre structure. La manière de décoder chaque type est définie dans la classe *Decoders* du programme. Il y actuellement 15 décodeurs implémentés.

3.2.3 ClassInfos

Les classinfos sont une information essentielle au décodage du fichier, en effet chaque fois qu'une entité est référencée, au lieu de l'obtenir par son nom qui peut être très long comme "CDOTA_Ability_Alchemist_UnstableConcoction" (48 bytes - 384 bits) le fichier va se servir d'un indentifiant de type nombre entier contenu entre 0 et 754 pour les fichiers récents donc seulement 10 bits (donc un gain de 374 bits à chaque référence de "CDOTA_Ability_Alchemist_UnstableConcoctionThrow").

Voici le format de ce message :

	ClassInfos
<pre> message CDemoClassInfo { message class_t { optional int32 class_id = 1; optional string network_name = 2; optional string table_name = 3; } repeated .CDemoClassInfo.class_t classes = 1; </pre>	

```
}

```

Il s'agit donc d'une série d'entrées contenant un indentifiant, un nom réseau et nom de table. L'identifiant est le nombre qui permettra de référencer l'entité par la suite. Le nom réseau est le nom complet de l'entité qui permet également de la retrouver dans les StringTables. Le nom de table ne sert à rien, il n'est d'ailleurs jamais défini pour aucune des entrées, il est possible que ce soit un reste d'une ancienne version du protocole où les deux noms étaient distinct.

Une fois les classinfos lues. Nous allons pouvoir passer au décodage des entités, cependant avant de pouvoir le faire il va falloir définir les propriétés des entités essentielles (celles qui étaient dans le String Table "instancebaseline").

Pour ce faire nous allons parcourir les valeur (champ *value*) de chacune des entrées de cette table et les décoder comme propriétés de cette entité, bien que l'entité à proprement parler n'existe pas encore.

3.2.4 Propriétés

La lecture des propriétés est essentielle, mais c'est également la plus compliquée. En effet afin de lire les propriétés d'une entité, il faut 3 choses, une flux de bits (non-aligné), un serialiseur (trouvé depuis les send tables) et un arbre d'Huffman contenant différentes opérations possible pour remplir les propriétés d'une entité.

Ainsi il faut se placer à la racine de l'arbre, puis, tant que la lecture n'est pas finie (tant que nous ne somme pas tombé sur une feuille contenant l'operation *FieldPathEncodeFinish*) alors si le prochain bit du flux est égal à 1, aller à l'enfant de droite, sinon à l'enfant de gauche.

Quand nous tombons sur une feuille, il faut exécuter l'opération de la feuille puis ajouter le champ à la liste des champ à décoder.

Une fois la liste des champs complète nous pouvons les décoder grâce aux sérialiseurs que nous avons trouvés plus tôt dans les send tables.

3.2.5 PacketEntities

Les données intégrées de type svc_PacketEntities forment le coeur du fichier. En effet dans ce format tout est une entité, du "monde" dans lequel le jeu se passe, aux héros contrôlés par les joueurs en passant par les équipes.

Voici la structure de ce type de message :

```
message CSVCMsg_PacketEntities {
    optional int32 max_entries = 1;
    optional int32 updated_entries = 2;
    optional bool is_delta = 3;
    optional bool update_baseline = 4;
    optional int32 baseline = 5;
    optional int32 delta_from = 6;
    optional bytes entity_data = 7;
    optional bool pending_full_frame = 8;
    optional uint32 active_spawn_group_handle = 9;
}
```

```
    optional uint32 max_spawnngroup_creationsequence = 10 ;  
}
```

Les données sur l'entité sont contenue dans le champ *entity_data* qui est de type bytes, il va donc falloir décoder de la manière suivante :

D'abord, lire l'identifiant de l'entité (numéro unique qui permet de l'identifier parmi toutes les entités existantes) encodé sur un Bit Var, la valeur que nous lisons est en réalité un delta qui permet de passer du dernier identifiant vers le prochain : $next = last + delta + 1$. Ensuite, 2 bits, appelons les x et y, qui vont déterminer l'opération à effectuer : création (x = 0 ; y = 1), mise à jour (x = 0 ; y = 0), suppression (x = 1 ; y = 1) ou retrait (x = 1 ; y = 0).

Dans le cas de la création, il faut :

1. Lire 10 bits pour trouver l'identifiant du type de l'entité, grâce auquel nous pouvons retrouver le type de l'entité et son serialiseur.
2. Lire 17 bits pour connaître le numéro de série de l'entité (pas utile dans notre cas)
3. Sauter 32 bits, leur utilité ne m'est pas connue jusqu'à présent
4. Lire les propriétés de l'entité comme décrit précédemment
5. Ajouter l'entité à notre liste d'entité

Dans le cas de la mise à jour, il faut :

1. Lire les propriétés de l'entité
2. Mettre à jour l'entité avec les nouvelles valeurs

Dans le cas de la suppression il suffit de supprimer l'entité de notre liste et dans le cas du retrait il n'y a rien à faire de particulier.

3.3 Implémentation