

## Κεφάλαιο 2: Θεωρητικό πλαίσιο

### 2.1 Εισαγωγή κεφαλαίου

Στο κεφάλαιο αυτό θα αναλυθεί το θεωρητικό πλαίσιο της εργασίας εξετάζοντας προσεκτικά τύπους serializers που χρησιμοποιούνται στην βιομηχανία, θα γίνει μία σύντομη επισκόπηση των συστημάτων serialization προς ανάλυση στο κεφάλαιο 3, καθώς και μια σειρά άλλων σχετικών πηγών. Επιπλέον, θα οροθετηθεί η έννοια των Foreign Function Interfaces, οι λειτουργίες τους άλλα και επικείμενα σοβαρά ζητήματα του σχεδιασμού τους. Επιπροσθέτως, θα αναλυθούν οι δύο βασικές κατηγορίες των wrapper και ποια η θεωρεία τους και ο ορισμός τους. Τέλος, θα γίνει μία ανάλυση των μηχανισμών serialization και saving συστημάτων που περιέχονται στις μηχανές δημιουργίας βιντεοπαιχνιδιών Unity και Unreal Engine.

### 2.2 Persistent Data στην Πληροφορική

Η «διατήρηση δεδομένων»(data preservation) στην Πληροφορική είναι η διαδικασία διατήρησης αρχείων και δεδομένων σε ένα στάδιο που τα καθιστά προσβάσιμα καθ' όλη τη διάρκεια του χρόνου. Τα δεδομένα αυτά θα πρέπει να αποθηκεύονται σε μορφές αρχείων που θα τα καθιστούν πιο χρήσιμα στο μέλλον, θα πρέπει να φυλάσσονται σε πολλές τοποθεσίες και τέλος να διατηρούνται σε ένα προστατευμένο «περιβάλλον» προκειμένου να διατηρηθούν(ref).

Συγκεκριμένα, τα μόνιμα δεδομένα(persistent data) αναφέρονται σε δεδομένα που διατηρούνται επ' αόριστον σε μη πτητικές συσκευές αποθήκευσης(non-volatile storage), όπως μαγνητικές ταινίες, σκληρούς δίσκους, Solid State Drives αλλά και οπτικούς δίσκους. Τα δεδομένα αυτά διατηρούνται ακόμη και μετά την απενεργοποίηση της συσκευής, δηλαδή χωρίς ρεύμα, από όπου προέρχεται και η ονομασία τους Non-Volatile Memory. Φυσικά, χρησιμοποιούνται πολλές τακτικές για να εξασφαλιστεί η προσβασιμότητα και η μακροπρόθεσμη διατήρησή τους. Χρησιμοποιούνται συστήματα αρχείων για την οργάνωση των δεδομένων(file systems), εφαρμόζονται τεχνικές δημιουργίας αντιγράφων ασφαλείας, όπως είναι τα διαφορικά(differential), τα αυξητικά(incremental) ή και τα πλήρη(complete) αντίγραφα ασφαλείας, τα οποία αποθηκεύονται εντός, εκτός είτε στο cloud, και γίνεται αντιγραφή των δεδομένων σε διάφορες συσκευές ή τοποθεσίες για λόγους ανθεκτικότητας. (ref)

### 2.3 Data Serialization

Στην υποενότητα αυτή γίνεται μία σύντομη ανασκόπηση στους ορισμούς του Serialization και Deserialization όπως αυτοί χρησιμοποιούνται στην Πληροφορική (εν. 2.3.1) και εξηγούνται οι βασικοί τύποι serialization όπως είναι οι XML, JSON, YAML και binary (εν. 2.3.2). Τέλος στην ενότητα 2.4, γίνεται μία επεξήγηση της διαχείρισης των address references από τη CPU και συγκεκριμένα της εξάλειψης των object references στη μνήμη του υπολογιστή κατά τη διαδικασία του serialization.

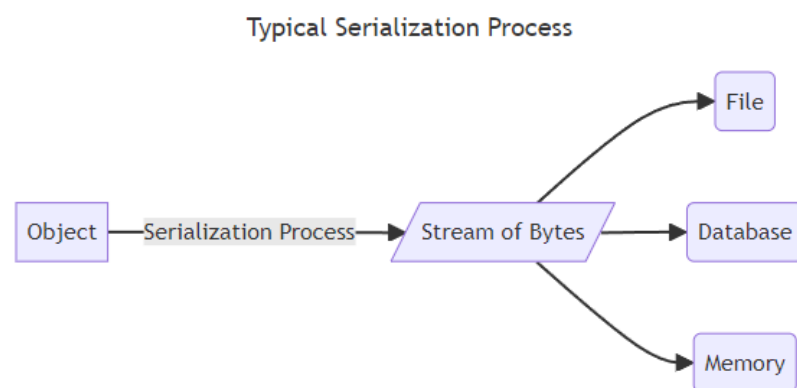
### 2.3.1 Ορισμός του Serialization και Deserialization

Στον κόσμο της Πληροφορικής, η διαδικασία της μετατροπής δεδομένων σε μία μορφή κατανοητή από έναν υπολογιστή ονομάζεται **Serialization**. Αναλυτικότερα, όπως διακρίνεται στην Εικόνα 1, η σειριοποίηση δεδομένων (data serialization) αναφέρεται στην πράξη του μετασχηματισμού περίπλοκων δομών δεδομένων ή καταστάσεων αντικειμένων (ref) από τη κατάσταση τους στη μνήμη σε έναν πίνακα από bytes ή σε μια μορφή που μπορεί εύκολα να καταγραφεί σε ένα αρχείο, να μεταφερθεί και να ξαναδημιουργηθεί αργότερα μέσω της αντίστροφης διαδικασίας, ονόματι **Deserialization**.

Συγκεκριμένα, το **Deserialization** είναι η διαδικασία μετάφρασης του αποθηκευμένου πίνακα ή αποθηκευμένης σειράς από bytes στην μορφή που ήταν πριν το **serialization**, κάτι που καθιστά τη διαδικασία του **deserialization** άμεσα συνδεδεμένη με τη διαδικασία του **serialization**. (ref)

Επιπλέον, είναι σημαντικό να αναφερθεί πως ο **Serializer** είναι κάτι διαφορετικό από το **Serialization Format** που εξάγει μετά τη διαδικασία του **serialization**. Το **serialization format** που εξάγει κάθε **serializer** βασίζεται σε συγκεκριμένη σύνταξη αναπαράστασης των **serialized** δεδομένων είτε αυτό είναι ένα **JSON string**, ένα **YAML** ή **XML structure** είτε είναι η **binary** αναπαράσταση αυτών. (ref)

Τέλος, κάθε **serializer** και με τη σειρά της η μορφή **serialized** δεδομένων που εξάγει έχει πλεονεκτήματα και εφαρμογές, όπου αυτά μπορούν να κυμαίνονται από το ποσοστό δυνατότητας ανθρώπινης ανάγνωσης μέχρι και το τελικό μέγεθος του παραχθέντος αρχείου για λόγους αποθήκευσης και μεταφοράς. (ref)



Εικόνα 1: Serialization process

### 2.3.2 Serialization formats

Η ακριβής σύνταξη και η δομή που χρησιμοποιείται για την αναπαράσταση των **serialized** δεδομένων καθορίζονται από τη μορφή του **serialization**. Οι μορφότυποι **serialization** εμφανίζουν ένα εύρος χαρακτηριστικών, όπως αναγνωσιμότητα, αποδοτικότητα και πολυπλοκότητα. Τα **JSON**, **YAML**, **XML** και **Message Pack (binary)** είναι μερικά παραδείγματα από αυτά τα μορφότυπα. Κάθε μορφή **serialization** περιγράφει κανόνες για την αναπαράσταση τύπων δεδομένων, την οργάνωση δομών δεδομένων και τον χειρισμό ειδικών περιπτώσεων όπως **nested objects** ή πίνακες, παρέχοντας έτσι μία σχεδίαση (schema) για το **serialization** δεδομένων σε διαφορετικά συστήματα και πλατφόρμες. (ref)

Τέλος, τα formats JSON, YAML, XML και binary διαθέτουν το καθένα μοναδικές ικανότητες προσαρμοσμένες σε συγκεκριμένες περιπτώσεις χρήσης, καλύπτοντας ποικίλες απαιτήσεις αναπαράστασης (data presentation), μετάδοσης (transmission) και αποθήκευσης δεδομένων (data preservation).

### 2.3.2a JSON Serialization

Ιστορικά, η μορφή JSON ή αλλιώς JavaScript Object Notation, είναι βασισμένη στο ανοικτό πρότυπο ECMA-262 3<sup>rd</sup> Edition 12/1999 της JavaScript (ref ECMA-262) από όπου και δημιουργήθηκε το «ECMA-404 The JSON Data Interchange Standard». Σκοπός της δημιουργίας του ήταν η μετατροπή των αναπαράστασεων δεδομένων σε μορφή απλού κειμένου που διαβάζονται από τον άνθρωπο σε αντικείμενα ECMAScript. Οι συμβολισμοί που χρησιμοποιεί είναι παρόμοιοι με εκείνους των κοινών γλωσσών προγραμματισμού όπως η C, η C++ και η Java και είναι εντελώς ανεξάρτητη από τις άλλες γλώσσες προγραμματισμού, έτσι αποτελεί μια κατάλληλη επιλογή για τη μετάδοση δεδομένων μεταξύ συστημάτων λόγω της αναγνωσιμότητας και της απλότητάς γραφής του (ref ECMA-404). Η ομαλή ενσωμάτωση καθίσταται δυνατή χάρη στον ελαφρύ πηγαίο σχεδιασμό του και την εγγενή υποστήριξη για την πλειονότητα των γλωσσών προγραμματισμού όπως προαναφέρθηκε. Το JSON λειτουργεί καλά σε καταστάσεις όπως τα διαδικτυακά API και τα αρχεία ρυθμίσεων, όπου είναι απαραίτητη η εύκολη ανάγνωση από τον άνθρωπο και η οργάνωση των δεδομένων.

Όπως φαίνεται στην Εικόνα 2, τα δεδομένα αποθηκεύονται σαν ζεύγη κλειδιών-τιμών (key-value pairs) και οι διαθέσιμοι τύποι δεδομένων προς αποθήκευση κυμαίνονται στα strings, στους integer/floating-point αριθμούς, στις Boolean τιμές, στις λίστες τακτοποιημένων τιμών (ordered lists), στις συλλογές μη τακτοποιημένων ζευγών κλειδιών-τιμών (unordered key-value collections) και τέλος, την αναπαράσταση της απουσίας τιμής μέσω του keyword null. (ref) Αξίζει να σημειωθεί πως οι συλλογές με μη τακτοποιημένα ζεύγη κλειδιών-τιμών (unordered key-value collections) και αυτά εσωτερικά μπορούν με τη σειρά τους να αναπαραστήσουν δεδομένα με τους προαναφερθέντες τύπους. (ref)

```
{ } sample.json > ...
1  {
2    "string_example": "Hello, world!",
3    "number_example": 42,
4    "float_example": 3.14,
5    "boolean_example": true,
6    "array_example": [1, 2, 3, 4, 5],
7    "object_example": {
8      "name": "John",
9      "age": 30,
10     "is_student": false
11   },
12   "null_example": null
13 }
```

Εικόνα 2: Δείγμα JSON αρχείου

### 2.3.2β XML Serialization

Η eXtensible Markup Language (XML) είναι μια απλή, πολύ ευέλικτη μορφή κειμένου που προέρχεται από την Standard Generalized Markup Language (SGML) του ISO 8879 (ref ISO 8879). Αρχικά σχεδιάστηκε από μία ομάδα του World Wide Web Consortium (W3C) το 1998 για να ανταποκριθεί στις προκλήσεις των μεγάλης κλίμακας ηλεκτρονικών εκδοτικών οίκων, όμως πλέον παίζει επίσης ολόένα και πιο σημαντικό ρόλο στην ανταλλαγή μιας μεγάλης ποικιλίας δεδομένων στο Ίντερνετ και αλλού. (ref from W3C).

Η XML, αναπαριστά τα δομημένα δεδομένα με ετικέτες (tags) που περικλείονται σε αγκύλες. Αυτές οι ετικέτες οριοθετούν τη δομή των δεδομένων και μπορούν να περιλαμβάνουν χαρακτηριστικά και nested στοιχεία (elements). Κατά το serialization, τα αντικείμενα ή οι δομές δεδομένων μετατρέπονται σε μία μορφή XML με βάση το επιλεγμένο schema. Αυτό περιλαμβάνει τη διερεύνηση των ιδιοτήτων (attributes) ή των πεδίων (fields) του αντικειμένου και τη δημιουργία των αντίστοιχων στοιχείων και χαρακτηριστικών XML για την ακριβή αναπαράσταση των δεδομένων. Οι δομές XML Schema Definition (XSD) και Document Type Definition (DTD) προσφέρουν επίσημες προδιαγραφές για τον ορισμό της δομής και των περιορισμών των εγγράφων που δημιουργούνται (ref). Επιπλέον, η XML επιτρέπει την δημιουργία custom δομών όπου μερικές από τις αλλαγές που μπορούν να επισημανθούν είναι το τελικό XML format που θα δημιουργηθεί, η διαχείριση ειδικών τύπων δεδομένων, ο καθορισμός συμβάσεων ονοματοδοσίας και η διαχείριση των namespaces. (ref) Σαν serialization format, η XML βρίσκει ευρεία χρήση στις υπηρεσίες ιστού για την ανταλλαγή δεδομένων μεταξύ πελατών και διακομιστών. Το πρωτόκολλο Simple Object Access Protocol (SOAP), για παράδειγμα, αξιοποιεί την XML για τη μορφοποίηση μηνυμάτων που μοιράζονται μεταξύ των κατανεμημένων του συστημάτων (distributed systems).

Τέλος, είναι άξιο να σημειωθεί πως η XML μοιράζεται ομοιότητες με την HTML, καθώς και οι δύο είναι γλώσσες σήμανσης (markup languages) που χρησιμοποιούνται για τη δόμηση και την οργάνωση του περιεχομένου. Και οι δύο χρησιμοποιούν ετικέτες (tags) που περικλείονται σε αγκύλες για να ορίσουν τα στοιχεία μέσα σε ένα έγγραφο. Ωστόσο, η XML διαφέρει από την HTML στο ότι είναι πιο ευέλικτη και επεκτάσιμη, επιτρέποντας τη δημιουργία προσαρμοσμένων ετικετών και δομών εγγράφων, προσαρμοσμένων σε συγκεκριμένες ανάγκες αναπαράστασης δεδομένων. Ενώ η HTML χρησιμοποιείται κυρίως για την εμφάνιση περιεχομένου στον ιστό, η XML χρησιμοποιείται για την αποθήκευση, τη μετάδοση και την ανταλλαγή δεδομένων σε διαφορετικά συστήματα και πλατφόρμες, επηρεασμένη από την προσέγγιση της HTML για τη σήμανση (markup) και τη δόμηση των δεδομένων. (ref)

Ως προς τους τύπους δεδομένων που υποστηρίζονται από την XML, όπως αυτά αναπαρίστανται στην Εικόνα 3, οι πιο βασικοί είναι το text το οποίο δέχεται από απλό κείμενο μέχρι και HTML markup μέσα σε ενότητες CDATA, αριθμούς όπως integers, floating-point αριθμούς, δεκαδικούς αριθμούς αλλά και επιστημονική σημειογραφία (scientific notation). Επιπλέον, υποστηρίζονται τιμές Boolean είτε σαν True-False είτε με 1 για True και 0 για False αλλά και η δυνατότητα αποθήκευσης δεδομένων σχετικά με την χρονολογία και την ώρα με βάση κάποιο format, όπως για παράδειγμα το "YYYY-MM-DD" για την χρονολογία (ISO 8601) (ref). Παρόλο που η XML είναι ένα format με βάση το κείμενο (text based) υποστηρίζει την αναπαράσταση δυαδικών δεδομένων (binary data) με τη χρήση τεχνικών όπως η κωδικοποίηση σε Base64 (ref) τα οποία και αποθηκεύονται σαν text μέσα στα XML στοιχεία.

Επιπροσθέτως, προσφέρει ιεραρχικές διατάξεις στοιχείων (elements) και χαρακτηριστικών (attributes), διευκολύνοντας τη δημιουργία περίπλοκων δομών δεδομένων. Ακόμη, παρέχει στους χρήστες τη δυνατότητα να προσαρμόζουν την αναπαράσταση των δεδομένων, ορίζοντας

προσαρμοσμένους τύπους δεδομένων με τη χρήση elements και attributes, ικανοποιώντας έτσι τις απαιτήσεις συγκεκριμένων τομέων.

Εν κατακλείδι, η XML ενισχύει την ολοκληρωμένη διαχείριση δεδομένων, επιτρέποντας τη συμπερίληψη μεταδεδομένων (metadata ή information about data) στα έγγραφα. Αυτά τα metadata περιλαμβάνουν κρίσιμες λεπτομέρειες όπως το συγγραφικό δικαίωμα του συγγραφέα, την ημερομηνία δημιουργίας, τις πληροφορίες έκδοσης και άλλα συναφή metadata, ενισχύοντας έτσι την κατανόηση του πλαισίου και τη διαχείριση των δεδομένων. (ref)

```
codeSamples > </> sample.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <data>
3      <!-- Textual data -->
4      <name>John Doe</name>
5
6      <!-- Numeric data -->
7      <age>30</age>
8
9      <!-- Boolean value -->
10     <is_student>true</is_student>
11
12     <!-- Date and time -->
13     <registration_date>2024-04-05T08:00:00</registration_date>
14
15     <!-- Binary data (Base64 encoded) -->
16     <profile_picture>base64_encoded_data_here</profile_picture>
17
18     <!-- Structured data -->
19     <address>
20         <street>123 Main St</street>
21         <city>New York</city>
22         <state>NY</state>
23         <zip_code>10001</zip_code>
24     </address>
25
26     <!-- Custom data type -->
27     <product>
28         <name>Laptop</name>
29         <price>999.99</price>
30         <manufacturer>XYZ Electronics</manufacturer>
31     </product>
32
33     <!-- Metadata -->
34     <metadata>
35         <author>John Smith</author>
36         <creation_date>2024-04-05</creation_date>
37         <version>1.0</version>
38     </metadata>
39 </data>
```

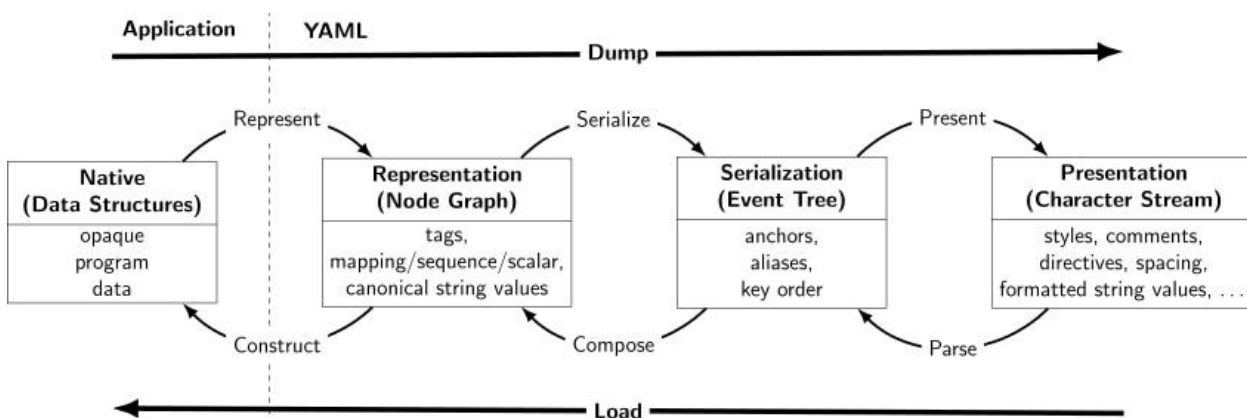
Εικόνα 3: Δείγμα XML αρχείου

### 2.3.2γ YAML Serialization

Η YAML Ain't Markup Language (YAML) είναι ένα serialization format δεδομένων που χρησιμοποιείται για την αναπαράσταση δομημένων δεδομένων (structured data). Λόγο της εύκολης ανάγνωσης του από τον άνθρωπο χρησιμοποιείται συχνά σε αρχεία ρυθμίσεων, στην ανταλλαγή δεδομένων μεταξύ προγραμμάτων και σε εφαρμογές όπου η αναγνωσιμότητα από τον άνθρωπο αποτελεί προτεραιότητα.

Ιστορικά, η YAML εμπνευσμένη από την XML, δημιουργήθηκε το 2001 από μία ομάδα ερευνητών με κύριους στόχους την εύκολη ανάγνωση της από τον άνθρωπο, την εύκολη μεταφορά και μετάδοση δεδομένων μεταξύ διαφόρων γλωσσών προγραμματισμού και την ευκολία χρήσης της. (ref) Παρόλο που η YAML δεν στηρίζεται σε κάποιο προ υπάρχων standard, όπως η JSON και η XML (ref), μερικά βασικά χαρακτηριστικά του σχεδιασμού της περιέχουν την προσπάθεια ομοιογένειας μεταξύ των τύπων δεδομένων της με τις εγγενείς δομές δεδομένων (native data structures) που έχουν οι δυναμικές γλώσσες προγραμματισμού (dynamic languages), τη σχεδίαση ενός συνεπής μοντέλου το οποίο θα υποστηρίζει γενικά εργαλεία (generic tools) και την έμφαση στην εκφραστικότητα και την επεκτασιμότητα της.

Τέλος, ο μηχανισμός επεξεργασίας της με βάση την επίσημη έρευνα σχεδιασμού της, όπως αυτή αναπαρίσταται στην Εικόνα 4, χαρακτηρίζεται σαν μηχανισμός “one-pass processing” (ref). Αυτό σημαίνει ότι ο αναλυτής (parser) της YAML διαβάζει τα δεδομένα μόνο μία φορά από την αρχή έως το τέλος και τα επεξεργάζεται καθώς τα συναντά. (ref)



Εικόνα 4: Επισκόπηση επεξεργασίας YAML (ref from book)

Επιγραμματικά, οι πιο βασικοί τύποι δεδομένων που υποστηρίζει η YAML ως προς την αποθήκευσή τους, όπως αυτά αναπαρίστανται στην Εικόνα 5, είναι οι κλίμακες (scalars) οι οποίες μπορούν να αποθηκεύσουν strings και multi-line strings, integers και floating-point αριθμούς, τιμές Boolean αλλά και την απουσία τιμής είτε με το keyword null ή με το σύμβολο ~. Επιπλέον, υποστηρίζονται οι διατεταγμένες συλλογές αντικειμένων (ordered item collections) ή όπως τις ονομάζει η YAML, τα Sequences όπου μπορούν να περιέχουν κάθε αναφερόμενο τύπο δεδομένων, οι συλλογές (Mappings) από ζεύγη κλειδιών-τιμών (key-value pairs) όπου το κλειδί πρέπει να είναι μοναδικό μέσα στο σύνολο του Mapping αλλά η τιμή μπορεί να περιέχει ακόμα και nested sequences ή άλλα mappings.

Τέλος, η YAML περιέχει και τα λεγόμενα Anchors και References. Συγκεκριμένα, το χαρακτηριστικό των anchors και των references που διακρίνεται στη YAML δεν αναφέρεται στα memory address references των αντικειμένων που μόλις έγιναν serialized αλλά χρησιμοποιούνται για τη δημιουργία ψευδώνυμων (aliases) στο ίδιο το YAML αρχείο. Μέσω της χρήσης τους, αποφεύγεται η διπλοτυπία διατύπωσης των δεδομένων μέσα στο αρχείο και αυξάνεται η ευκολία ανάγνωσης του.

Όταν ορίζεται ένα anchor και μετά γίνεται reference αυτού σε κάποιο σημείο μέσα στο YAML αρχείο, ουσιαστικά δίνεται η εντολή στον parser να χειριστεί αυτές τις εμφανίσεις σαν πανομοιότυπες ως προς το περιεχόμενο τους. Εν γένει, είναι ένας μηχανισμός για την επαναχρησιμοποίηση δομών δεδομένων εντός του ίδιου αρχείου YAML, αλλά δεν δημιουργεί καμία σύνδεση με τα αρχικά serialized αντικείμενα ή δομές δεδομένων εκτός του πλαισίου του αρχείου. (ref)

```
sample.yaml
1  # Scalars
2  name: John Doe
3  age: 30
4  is_student: false
5  null_value: null
6
7  # Sequences
8  hobbies:
9    - Reading
10   - Hiking
11   - Cooking
12
13  # Mappings
14  address:
15    city: New York
16    street: 123 Main St
17    zip_code: "10001"
18
19  # Anchors and References
20  person1: &person
21    name: Alice
22    age: 25
23
24  #Reference to anchor &person
25  person2: *person
26
```

Εικόνα 5: Δείγμα αρχείου YAML

### 2.3.2δ Binary Serialization

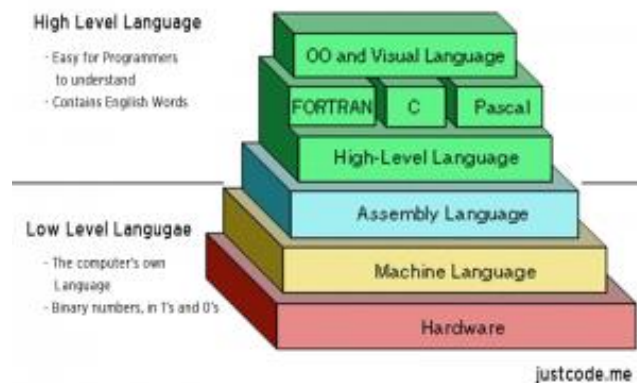
Η δυαδική σειριοποίηση (binary serialization) είναι μια διαδικασία μετατροπής δομών δεδομένων ή αντικειμένων σε δυαδική μορφή (binary), ώστε να μπορούν να αποθηκευτούν, να μεταδοθούν ή να ανακατασκευαστούν αποτελεσματικά αργότερα. Σε αντίθεση με τις μορφές JSON ή XML και YAML, οι οποίες είναι δυνατό να αναγνωστούν από έναν άνθρωπο, το binary serialization κωδικοποιεί τα δεδομένα σε μια συμπαγή, αναγνώσιμη μόνο από μηχανήματα μορφή, η οποία είναι ιδιαίτερα χρήσιμη



σε σενάρια όπου η αποδοτικότητα, η ταχύτητα, το τελικό μέγεθος των serialized δεδομένων ή η μειωμένη χρήση bandwidth είναι σημαντικά ζητήματα.

Αρχικά, σε αντίθεση με τα προαναφερθέντα serialization formats και serializers, το binary serialization ως έννοια, υπάρχει εγγενώς λόγω της φύσης των υπολογιστών που χειρίζονται δυαδικά δεδομένα. Από τις πρώτες ημέρες της πληροφορικής, οι προγραμματιστές χρειαζόνταν τρόπους αποθήκευσης και μετάδοσης δεδομένων σε μορφή που οι υπολογιστές μπορούσαν να κατανοήσουν αποτελεσματικά. Οι ρίζες του binary serialization μπορούν να εντοπιστούν στην πρώιμη ανάπτυξη των συστημάτων υπολογιστών και των γλωσσών προγραμματισμού (ref). Καθώς οι υπολογιστές εξελίσσονταν και γίνονταν πιο ισχυροί, οι προγραμματιστές επινόησαν μεθόδους για την αναπαράσταση δεδομένων σε δυαδική μορφή για τη βελτιστοποίηση του χώρου αποθήκευσης, τη βελτίωση της απόδοσης και τη διευκόλυνση της επικοινωνίας μεταξύ διαφορετικών συστημάτων. Στις αρχές της πληροφορικής, το binary serialization ήταν συχνά χειροκίνητα κωδικοποιημένο, με τους προγραμματιστές να κωδικοποιούν και να αποκωδικοποιούν χειροκίνητα δομές δεδομένων σε δυαδική μορφή χρησιμοποιώντας τεχνικές χαμηλού επιπέδου (low-level), όπως bit-manipulation και πράξεις σε επίπεδο byte (ref).

Καθώς οι γλώσσες προγραμματισμού και οι πρακτικές ανάπτυξης λογισμικού ωρίμαζαν, αναπτύχθηκαν αφαιρετικές (abstractions) μέθοδοι και βιβλιοθήκες υψηλότερου επιπέδου (high-level) για το binary serialization για την απλούστευση της διαδικασίας και τη βελτίωση της παραγωγικότητας. Στην Εικόνα 6 διακρίνονται οι έννοιες low-level και high-level.



Εικόνα 6: Low-Level και High-Level έννοιες σε γράφημα (ref)

Ισχυρά παραδείγματα είναι οι γλώσσες C και Pascal, οι οποίες περιείχαν low-level εντολές για την ανάγνωση και γραφή δυαδικών δεδομένων σε αρχεία, ενώ μεταγενέστερες γλώσσες όπως η Java και η C# εισήγαγαν ενσωματωμένα frameworks για serialization όπως είναι το ObjectOutputStream και ObjectInputStream (ref) και ο BinaryFormatter (ref) αντίστοιχα, για την αυτοματοποίηση της διαδικασίας του serialization και τον χειρισμό πολύπλοκων δομών αντικειμένων.

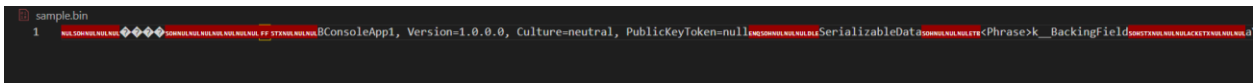
Επιπροσθέτως, με βάση αυτές τις αρχές οι binary serializers που έχουν δημιουργηθεί μπορούν να υποστηρίξουν ένα μεγάλο εύρος τύπων δεδομένων όπως (ref):

- **Primitive** τύπους: Οι binary serializers μπορούν να χειριστούν Primitive τύπους δεδομένων, όπως ακέραιους αριθμούς (π.χ. int, long, short), αριθμούς κινητής υποδιαστολής (π.χ. float, double), χαρακτήρες (π.χ. char) και τιμές Boolean.
- **Composite** τύπους: Οι binary serializers μπορούν να χειριστούν Composite τύπους δεδομένων που αποτελούνται από πολλούς primitive τύπους. Σε αυτούς περιλαμβάνονται πίνακες, λίστες, maps (ή dictionaries), sets, tuples και άλλοι collection τύποι.



- **Custom Objects:** Προσαρμοσμένα αντικείμενα ή κλάσεις, οι οποίες μπορεί να περιέχουν ένα συνδυασμό πρωτόγονων τύπων, σύνθετων τύπων και references σε άλλα αντικείμενα μπορούν επίσης να γίνουν serialized από έναν binary serializer. Οι βιβλιοθήκες serialization συχνά παρέχουν μηχανισμούς για την προσαρμογή της διαδικασίας serialization αυτών των αντικειμένων είτε μέσω κάποιου configuration αρχείου ή κάποιου custom object schema που θα δημιουργήσει ο προγραμματιστής.
- **Nullable** τύπους: Οι nullable τύποι ή οι τιμές στις οποίες μπορεί να ανατεθεί είτε ένα null reference είτε μια έγκυρη τιμή του υποκείμενου τύπου, διαχειρίζονται επίσης από τους binary serializers.
- **Strings:** Τα strings (πίνακες από char) είναι μία ιδιαίτερη περίπτωση όσον αφορά το binary serialization. Λόγο των διαφορετικών κωδικοποιήσεων όπως είναι οι UTF-8 και UTF-16 - μαζί με μία πληθώρα πολλών άλλων- συχνά οι binary serialization βιβλιοθήκες περιέχουν μηχανισμούς για τη διαχείριση του μεγέθους των string αλλά και υποκείμενα προβλήματα σχετικά με την επιλεγμένη κωδικοποίηση.

Το τελικό παραχθέν αρχείο, όπως διακρίνεται στην Εικόνα 7, δεν αποτελεί μία μορφή η οποία μπορεί να διαβαστεί από τον άνθρωπο και έτσι εάν κάποιος επιχειρήσει να το ανοίξει με κάποιον text editor όπως Notepad ή Notepad++ (ref from site) θα διακρίνει μία αλληλουχία γραμμάτων, συμβόλων, αριθμών ή και ακόμα μη κατανοητά από τον άνθρωπο σύμβολα. Αυτό το φαινόμενο συμβαίνει διότι το παραχθέν αρχείο περιέχει raw binary δεδομένα ή αλλιώς machine code (ref) τα οποία και είναι κατανοητά μόνο από έναν υπολογιστή. Με αυτό τον τρόπο παρέχεται και ένα μικρό επίπεδο ασφάλειας ως προς την ανάγνωση και τροποποίηση των δεδομένων, χωρίς όμως αυτό να αποτελεί κάποιο επίπεδο encryption (ref).



Εικόνα 7: Δείγμα binary αρχείου

Τέλος, στην Εικόνα 8, φαίνεται η αναπαράσταση του δείγματος στην Εικόνα 7 μέσω του προγράμματος HxD32 (ref) που με βάση την επίσημη ιστοσελίδα του περιγράφεται ως:

*«HxD is a carefully designed and fast hex editor which, additionally to raw disk editing and modifying of main memory (RAM), handles files of any size.»(ref)*

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	00	00	00	FF	FF	FF	FF	01	00	00	00	00	00	00
00000010	00	0C	02	00	00	00	42	43	6F	6E	73	6F	6C	65	41	70
00000020	70	31	2C	20	56	65	72	73	69	6F	6E	3D	31	2E	30	2E
00000030	30	2E	30	2C	20	43	75	6C	74	75	72	65	3D	6E	65	75
00000040	74	72	61	6C	2C	20	50	75	62	6C	69	63	4B	65	79	54
00000050	6F	6B	65	6E	3D	6E	75	6C	6C	05	01	00	00	00	10	53
00000060	65	72	69	61	6C	69	7A	61	62	6C	65	44	61	74	61	01
00000070	00	00	00	17	3C	50	68	72	61	73	65	3E	6B	5F	5F	42
00000080	61	63	6B	69	6E	67	46	69	65	6C	64	01	02	00	00	00
00000090	06	03	00	00	00	61	59	6F	75	20	61	63	74	75	61	6C
000000A0	6C	79	20	64	65	73	65	72	69	61	6C	69	7A	65	64	20
000000B0	6D	79	20	6D	61	6A	6F	72	20	73	61	6D	70	6C	65	20
000000C0	66	69	6C	65	2E	20	43	6F	6E	67	72	61	74	75	6C	61
000000D0	74	69	6F	6E	73	20	74	6F	20	79	6F	75	2C	20	62	79
000000E0	20	4D	41	44	20	66	6F	72	20	53	41	45	20	77	69	74
000000F0	68	20	6C	6F	76	65	21	0B								

Εικόνα 8: Inspection του αρχείου sample.bin με HxD32

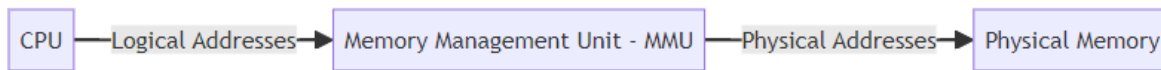
## 2.4 Memory management, addresses και serialization

Στην ενότητα αυτή γίνεται μία σύντομη ανάλυση των τρόπων με τους οποίους ένα υπολογιστικό σύστημα διαχειρίζεται τη μνήμη που του παρέχεται. Γίνεται μία αναφορά στις λογικές και φυσικές διευθύνσεις που παράγει ο επεξεργαστής, τις λειτουργίες διαχείρισης μνήμης και εικονικής μνήμης και τέλος εξηγείται τι συμβαίνει σε αυτές τις διευθύνσεις κατά τη διαδικασία του serialization.

### 2.4.1 Logical και Physical addresses

Στον τομέα των υπολογιστικών, οι λογικές διευθύνσεις (logical addresses) και οι φυσικές διευθύνσεις (physical addresses) έχουν έναν σημαντικό ρόλο στη διαδικασία διαχείρισης της μνήμης.

Οι λογικές διευθύνσεις είναι εικονικές διευθύνσεις (virtual addresses) που δημιουργούνται από την CPU και αντιπροσωπεύουν θέσεις στο λογικό χώρο διευθύνσεων που είναι διαθέσιμες και προσβάσιμες σε ένα πρόγραμμα. Αυτές οι διευθύνσεις στην πραγματικότητα, αποτελούν μία αφηρημένη (abstracted) και ανεξάρτητη έννοια από την πραγματική διαμόρφωση της φυσικής μνήμης, παρέχοντας ένα σταθερό περιβάλλον διεπαφής για την αλληλεπίδραση των προγραμμάτων με τη μνήμη (ref). Από την άλλη πλευρά, οι φυσικές διευθύνσεις αναφέρονται στις φυσικές θέσεις στο hardware μνήμης του υπολογιστή, όπου αποθηκεύονται τα δεδομένα. Αντιστοιχούν άμεσα στα πραγματικά memory cells του hardware, καθορίζοντας τις πραγματικές θέσεις στις οποίες βρίσκονται τα δεδομένα. Η μετάφραση μεταξύ των λογικών διευθύνσεων και των φυσικών διευθύνσεων διαχειρίζεται από τη μονάδα διαχείρισης μνήμης (Memory Management Unit - MMU) του λειτουργικού συστήματος, η οποία αντιστοιχίζει τις λογικές διευθύνσεις στις αντίστοιχες φυσικές διευθύνσεις, διευκολύνοντας την αποτελεσματική πρόσβαση και διαχείριση της μνήμης. Αυτή η διαδικασία αναπαρίσταται στην Εικόνα 9. Αυτό το επίπεδο αφαίρεσης μεταξύ λογικών και φυσικών διευθύνσεων επιτρέπει την ευέλικτη κατανομή, προστασία και αργότερα, virtualization της μνήμης στα σύγχρονα συστήματα υπολογιστών (ref).



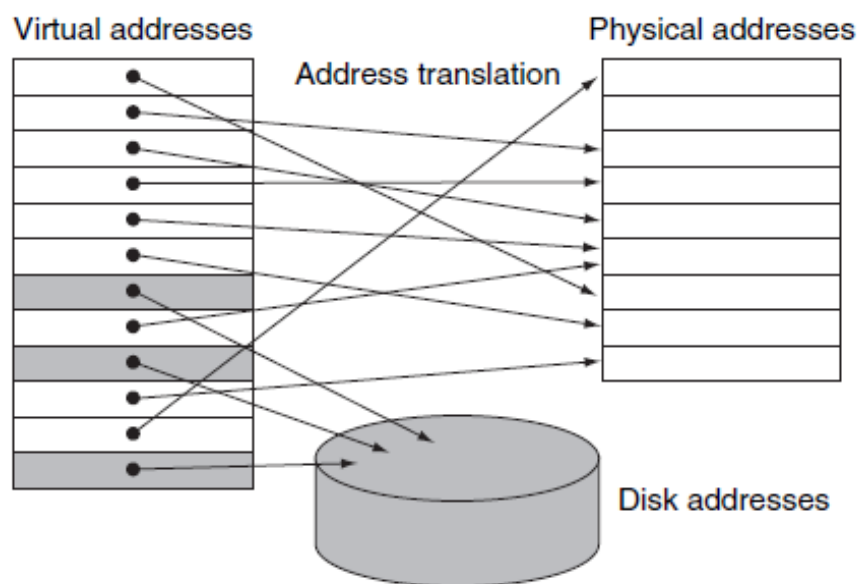
Εικόνα 9: Address binding on the MMU (ref from book 2022)

## 2.4.2 Memory Management και Virtual Memory σε ένα λειτουργικό σύστημα

Η διαχείριση της μνήμης σε ένα λειτουργικό σύστημα περιλαμβάνει την αποτελεσματική κατανομή και χρήση της μνήμης για την υποστήριξη των διεργασιών που εκτελούνται. Αυτό επιτυγχάνεται μέσω διαφόρων τεχνικών και schemes. Δύο βασικές τεχνικές που χρησιμοποιούνται στη διαχείριση της μνήμης είναι το paging και το segmentation. Το paging χωρίζει τη φυσική μνήμη σε μπλοκ σταθερού μεγέθους που ονομάζονται frames και τη λογική μνήμη σε μπλοκ σταθερού μεγέθους που ονομάζονται pages, επιτρέποντας την ευέλικτη κατανομή μνήμης και την αποτελεσματική χρήση της φυσικής μνήμης (ref). Όταν μια διεργασία (process) ζητά μνήμη, το λειτουργικό σύστημα κατανέμει ένα ή περισσότερα pages, τα οποία διαχειρίζεται μέσω ενός πίνακα σελίδων (page table). Εάν το ζητούμενο page δεν βρίσκεται στην κύρια μνήμη – όπως η RAM και η ROM - εμφανίζεται το λεγόμενο page fault, προτρέποντας το λειτουργικό σύστημα να φέρει το page από τον δευτερεύοντα αποθηκευτικό χώρο – όπως HDDs, SSDs και παρόμοια αποθηκευτικά μέσα - στη μνήμη.

Το segmentation, από την άλλη πλευρά, διαιρεί τη λογική μνήμη σε τμήματα μεταβλητού μεγέθους, που αντιπροσωπεύουν διαφορετικά τμήματα ενός προγράμματος (ref). Τα τμήματα μπορεί να περιλαμβάνουν κώδικα, δεδομένα, stack και heap. Ενώ το segmentation προσφέρει ευελιξία, εισάγει προκλήσεις όπως το λεγόμενο fragmentation. Το external fragmentation, όπως αποκαλείται, προκύπτει όταν η ελεύθερη μνήμη διαιρείται σε μικρά, μη συνεχόμενα μπλοκ, ενώ το internal fragmentation εμφανίζεται όταν τα τμήματα είναι μεγαλύτερα από τα απαραίτητα, οδηγώντας σε σπατάλη χώρου εντός των τμημάτων. Για την αντιμετώπιση αυτών των προκλήσεων, τα σύγχρονα λειτουργικά συστήματα χρησιμοποιούν συχνά έναν συνδυασμό paging και segmentation, γνωστό ως segmented paging, για να επιτύχουν αποτελεσματική διαχείριση της μνήμης. (ref)

Η εικονική μνήμη (virtual memory), όπως αυτή αναπαρίσταται στην Εικόνα 10, είναι μια θεμελιώδης έννοια στα λειτουργικά συστήματα που επιτρέπει την αποτελεσματική διαχείριση της μνήμης με την επέκταση της διαθέσιμης φυσικής μνήμης μέσω της χρήσης δευτερεύουσας αποθήκευσης, συνήθως ενός HDD ή ενός SSD. Στον πυρήνα της, η εικονική μνήμη επιτρέπει την εκτέλεση διεργασιών που ενδέχεται να μην χωρούν εξ ολοκλήρου στη διαθέσιμη φυσική μνήμη. Αντί να απαιτείται η ταυτόχρονη φόρτωση όλων των τμημάτων ενός προγράμματος στη μνήμη RAM, η εικονική μνήμη διαιρεί το χώρο διευθύνσεων μιας διεργασίας σε μικρότερες μονάδες που ονομάζονται pages ή segments. Αυτές οι μονάδες αντιστοιχίζονται στη συνέχεια δυναμικά μεταξύ της κύριας μνήμης και του δευτερεύοντος αποθηκευτικού χώρου από το λειτουργικό σύστημα μέσω μίας διαδικασίας ονόματι memory mapping, επιτρέποντας στη CPU να έχει πρόσβαση στα δεδομένα ανάλογα με τις ανάγκες της. Αυτή η προσέγγιση επιτρέπει την ψευδαίσθηση ενός τεράστιου και συνεχούς χώρου μνήμης, ακόμη και όταν η φυσική μνήμη είναι περιορισμένη, με τη άμεση ανταλλαγή δεδομένων μεταξύ της RAM και του δίσκου, όπως απαιτείται. Η εικονική μνήμη παίζει καθοριστικό ρόλο σε multitasking περιβάλλοντα, επιτρέποντας την ταυτόχρονη εκτέλεση πολλαπλών διεργασιών, βελτιστοποιώντας παράλληλα τη χρήση της μνήμης και την απόδοσή της.



Εικόνα 10: Virtual memory representation (ref from book 2022)

### 2.4.3 Τα memory addresses κατά το serialization

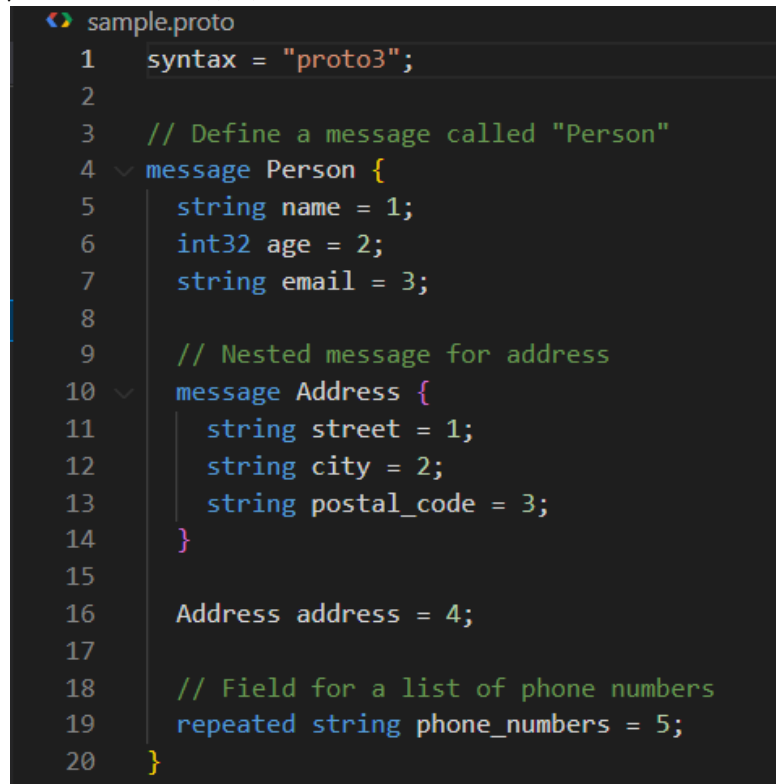
Όπως αναλύθηκε στις προηγούμενες ενότητες, οι διευθύνσεις μνήμης είναι συγκεκριμένες για το περιβάλλον εκτέλεσης στο οποίο εκτελείται ένα πρόγραμμα και δεν έχουν νόημα εκτός αυτού λόγω της παροδικής τους φύσης. Σε ένα εκτελούμενο πρόγραμμα, οι διευθύνσεις μνήμης εκχωρούνται δυναμικά από το MMU του λειτουργικού συστήματος καθώς οι δομές δεδομένων και οι μεταβλητές κατανέμονται στη μνήμη. Αυτές οι διευθύνσεις είναι σχετικές με το χώρο μνήμης του προγράμματος και μπορεί να αλλάζουν κάθε φορά που εκτελείται το πρόγραμμα ή ακόμη και κατά τη διάρκεια της εκτέλεσης του, καθώς η μνήμη κατανέμεται και αποδεσμεύεται δυναμικά. Κατά συνέπεια, οι διευθύνσεις μνήμης στερούνται της φορητότητας και δεν μπορούν να βασιστούν για μόνιμη αποθήκευση ή επικοινωνία μεταξύ διαφορετικών εκτελέσεων ενός προγράμματος ή μεταξύ διαφορετικών συστημάτων. Αντ' αυτού, οι διαδικασίες serialization, όπως προ αναλύθηκε στην ενότητα 2.3, επικεντρώνονται στην καταγραφή της κατάστασης των δεδομένων ενός αντικειμένου, όπως οι τιμές και η δομή του, η οποία μπορεί να ανακατασκευαστεί ανεξάρτητα από συγκεκριμένες διευθύνσεις μνήμης όταν γίνει deserialize σε άλλο περιβάλλον εκτέλεσης ή σύστημα.

## 2.5 Επισκόπηση Serializers προς ανάλυση

Στην ενότητα αυτή θα γίνει μία σχολαστική εξέταση των μεθόδων serialization δεδομένων όπου αποτελούν επιτακτική ανάγκη για τη βελτιστοποίηση της χρήσης των πόρων και τη βελτίωση της συνολικής απόδοσης ενός βιντεοπαιχνιδιού. Στο πλαίσιο αυτό, η επιλογή των κατάλληλων serializers αποκτά ύψιστη σημασία, ιδίως κατά την ανάπτυξη συστημάτων αποθήκευσης, τα οποία απαιτούν γρήγορους και αποτελεσματικούς μηχανισμούς αποθήκευσης και ανάκτησης δεδομένων. Μεταξύ της πληθώρας των διαθέσιμων επιλογών serialization, το MessagePack και το Protocol Buffers (Protobuf) αναδεικνύονται για τη συμπαγή δομή, την ταχύτητα και την ευελιξία τους (ref).

## 2.5.1 Protocol Buffers

Το Protocol Buffers (Protobuf) είναι μια μέθοδος serialization δομημένων δεδομένων, που αναπτύχθηκε από την Google, κάτι που τον καθιστά ιδιαίτερα δημοφιλή αλλά και σταθερό. Έχει σχεδιαστεί για να είναι ένας γρήγορος, αποδοτικός και γλωσσικά ουδέτερος μηχανισμός serialization δομημένων δεδομένων, καθιστώντας τον ιδανικό για πρωτόκολλα επικοινωνίας, αποθήκευση δεδομένων και συστήματα RPC (Remote Procedure Call). Αυτό το επιτυγχάνει με την δημιουργία των λεγόμενων “messages”, όπως αυτή φαίνεται στην Εικόνα 11, όπου και παίζουν κομβικό ρόλο στην συνολική λειτουργία του serializer (ref).



```
sample.proto
1  syntax = "proto3";
2
3  // Define a message called "Person"
4  message Person {
5      string name = 1;
6      int32 age = 2;
7      string email = 3;
8
9      // Nested message for address
10     message Address {
11         string street = 1;
12         string city = 2;
13         string postal_code = 3;
14     }
15
16     Address address = 4;
17
18     // Field for a list of phone numbers
19     repeated string phone_numbers = 5;
20 }
```

Εικόνα 11: Δείγμα .proto message αρχείου

Τα βασικά χαρακτηριστικά που επιλέχθηκε το Protobuf προς ανάλυση είναι τα παρακάτω:

- **Αποδοτικό serialization:** Το Protobuf χρησιμοποιεί μια binary μορφή serialization, η οποία είναι πιο συμπαγής και αποδοτική σε σύγκριση με άλλες μορφές όπως η XML, YAML ή η JSON. Αυτό έχει ως αποτέλεσμα - όπως και αναλύθηκε στην προηγούμενη ενότητα - μικρότερα μεγέθη αρχείων μειώνοντας έτσι τις απαιτήσεις αποθήκευσης.
- **Schema Definition Language:** Το Protobuf χρησιμοποιεί μία language-agnostic γλώσσα δημιουργίας schemas για να ορίσει τη δομή των δεδομένων που γίνονται serialize. Αυτή η γλώσσα δημιουργίας schema επιτρέπει τον ορισμό πεδίων, τους τύπους αυτών και την σειρά τους μέσα στα “messages”.

- **Data Streaming:** Το Protobuf υποστηρίζει data streaming, επιτρέποντας στις εφαρμογές να επεξεργάζονται αποτελεσματικά μεγάλα σύνολα δεδομένων χωρίς να φορτώνουν όλα τα δεδομένα στη μνήμη ταυτόχρονα.
- **Υποστήριξη πολλαπλών γλωσσών:** Υπάρχει υποστήριξη για πολλές γλώσσες προγραμματισμού, όπως C++, Java, Python, C# και πολλές άλλες. Αυτό επιτρέπει τον ορισμό των δομών δεδομένων μία φορά στα προ αναφερθέντα αρχεία .proto και την χρήση τους σε διαφορετικές πλατφόρμες και γλώσσες.
- **Δημιουργία κώδικα:** Το Protobuf χρησιμοποιεί μια δυναμική τεχνική δημιουργίας κώδικα για τη δημιουργία κλάσεων μίας συγκεκριμένης γλώσσας για το serialization και το deserialization των “messages” που το αφορούν. Αυτός ο παραγόμενος κώδικας είναι ιδιαίτερα βελτιστοποιημένος για απόδοση και παρέχει ασφάλεια στους τύπους των δεδομένων.
- **Επεκτασιμότητα:** Με την δυνατότητα επέκτασης των ήδη υπάρχοντων “messages”, μπορούν εύκολα να προστεθούν πεδία σε αυτά χωρίς την τροποποίηση του κυρίου schema του αρχικού “message”. Αυτό είναι κάτι ιδιαίτερα χρήσιμο σε συστήματα που χρησιμοποιούν πολλαπλούς τύπους δεδομένων ή όταν γίνεται ενσωμάτωση τρίτων πακέτων σε μία εφαρμογή, κάτι που συμβαίνει διαρκώς στον τομέα των βιντεοπαιχνιδιών.

Συνοψίζοντας, το Protobuf προσφέρει έναν γρήγορο, αποτελεσματικό και γλωσσικά ουδέτερο τρόπο serialization δομημένων δεδομένων, με υποστήριξη πολλαπλών γλωσσών προγραμματισμού, αποτελεσματική σειριοποίηση, backwards compatibility και ενσωμάτωση με RPC frameworks (ref). Αυτό τον καθιστά μία πολύ καλή επιλογή για τη δημιουργία κατανεμημένων συστημάτων και πρωτοκόλλων επικοινωνίας, ιδιαίτερα σε εφαρμογές με κρίσιμες επιδόσεις, όπως υπηρεσίες ιστού μεγάλης κλίμακας ή βιντεοπαιχνίδια.

## 2.5.2 MsgPack C++

Το MessagePack είναι μία open-source binary μέθοδος serialization σχεδιασμένη με στόχο την αποδοτικότητα. Χρησιμοποιείται συχνά σε πρωτόκολλα επικοινωνίας και αποθήκευσης δεδομένων, όπου το μέγεθος και η ταχύτητα είναι κρίσιμα (ref thesis 2022). Και το MessagePack χρησιμοποιεί structs ή κλάσεις γραμμένα στην εκάστοτε γλώσσα σαν data containers για το serialization των δεδομένων, όπως αυτό παρουσιάζεται στην Εικόνα 12.

```

C++ sample.cpp > ...
1  #include <msgpack.hpp>
2  #include <iostream>
3  #include <vector>
4
5  struct Person {
6      std::string name;
7      int age;
8      std::vector<std::string> hobbies;
9
10     // MessagePack serialization method
11     MSGPACK_DEFINE(name, age, hobbies);
12 };

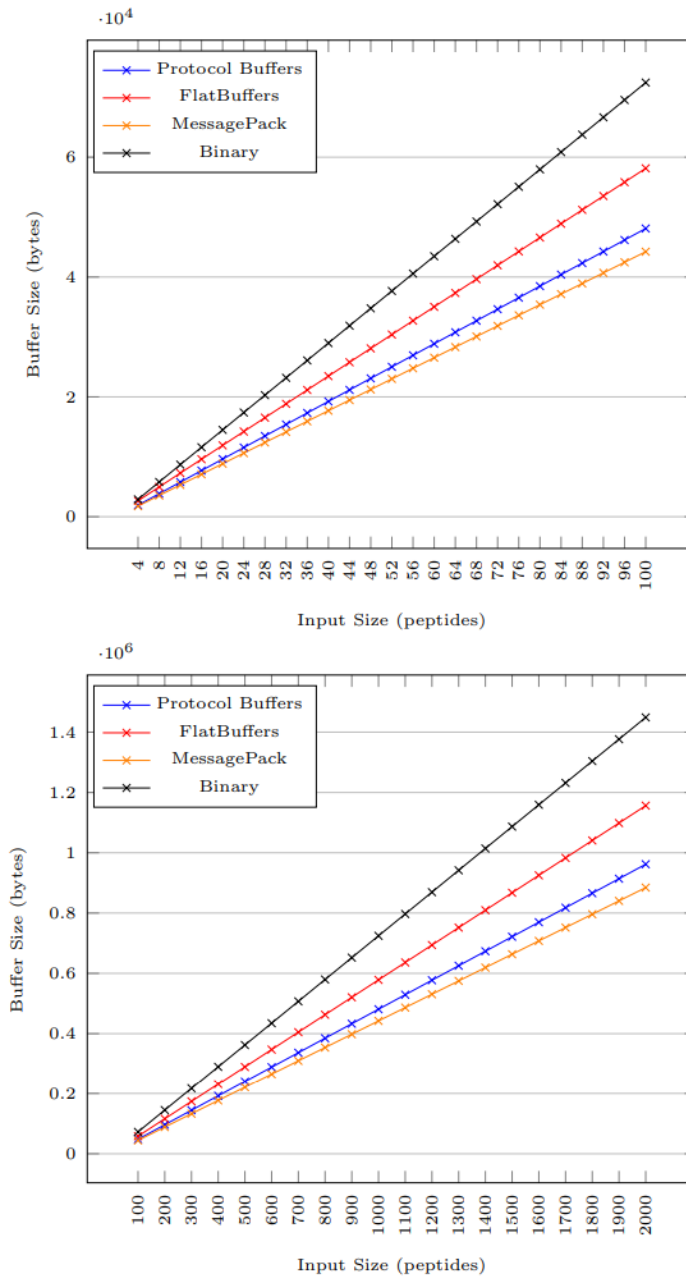
```

Εικόνα 12: Δείγμα MessagePack struct προς serialization σε C++

Τα βασικά χαρακτηριστικά που επιλέχθηκε το MessagePack προς ανάλυση είναι τα παρακάτω:

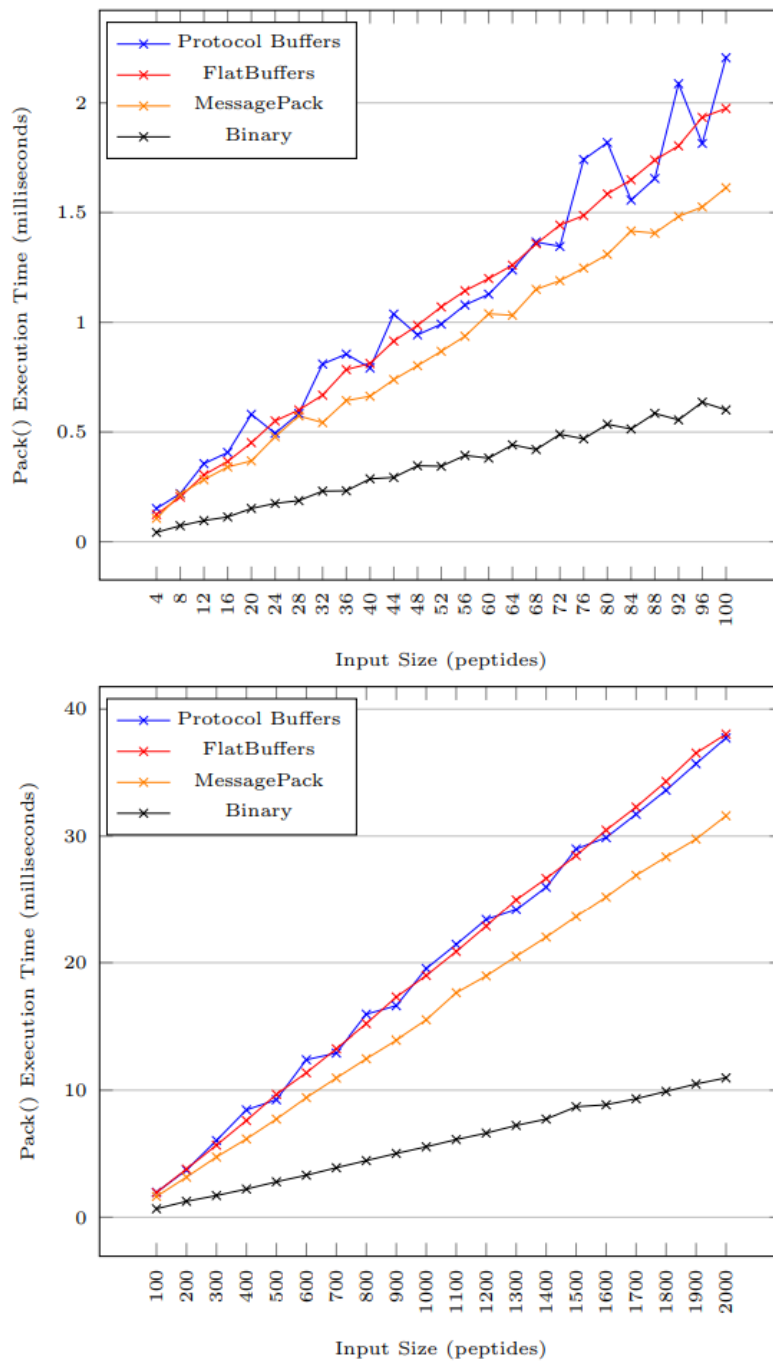
- **Συμπαγής binary μορφή:** Το MessagePack αναπαριστά τα δεδομένα σε δυαδική μορφή, που σύμφωνα με την επίσημη ιστοσελίδα του αλλά και μία έρευνα του University of Houston το 2022 (ref), φαίνεται πως είναι πιο συμπαγής σε σύγκριση με τις μορφές κειμένου όπως το JSON, XML και YAML ακόμα και σε σύγκριση με το Protobuf, όπως διακρίνεται στην Εικόνα 13 (ref). Αυτή η συμπαγής μορφή μειώνει το μέγεθος των μεταδιδόμενων δεδομένων, καθιστώντας τα αποδοτικά για επικοινωνία και αποθήκευση στο δίκτυο.





Εικόνα 13: Μεγέθη serialized δεδομένων, με τα μικρά εισαγόμενα μεγέθη (πάνω) και τα μεγάλα (κάτω) (ref from Casey 2022)

- **Αποδοτικό serialization και deserialization:** Με βάση μία έρευνα του University of Houston το 2022 (ref), το MessagePack έχει έναν πολύ αποτελεσματικό αλγόριθμο serialization και deserialization, όπως τα αποτελέσματα του φαίνονται στην Εικόνα 14. Αυτή η αποδοτικότητα είναι ιδιαίτερα επωφελής στη C++ όπου η απόδοση είναι κρίσιμη.



Εικόνα 14:Χρόνος εκτέλεσης της μεθόδου Pack() για όλους τους μηχανισμούς, με μικρότερες εισόδους (πάνω) και μεγαλύτερες εισόδους (κάτω)(ref from Casey 2022)

- **Language agnostic:** Αυτό σημαίνει ότι μπορεί να χρησιμοποιηθεί σε διάφορες γλώσσες προγραμματισμού χωρίς προβλήματα συμβατότητας λόγω της φύσης που έχει με το JSON format. Αυτό το καθιστά κατάλληλο για ετερογενή περιβάλλοντα όπου χρησιμοποιούνται πολλές γλώσσες για την ανάπτυξη μίας εφαρμογής.

- **Υποστήριξη τύπων δεδομένων:** Υπάρχει υποστήριξη για integers, floating-point numbers, συμβολοσειρές, πίνακες και maps.
- **Data Streaming:** Το MessagePack υποστηρίζει data streaming, επιτρέποντας στις εφαρμογές να επεξεργάζονται αποτελεσματικά μεγάλα σύνολα δεδομένων χωρίς να φορτώνουν όλα τα δεδομένα στη μνήμη ταυτόχρονα.
- **Open Source:** Είναι open source (ref from site) κάτι που επιτρέπει στον εκάστοτε developer να σμιλέψει όλες τις διαδικασίες του όπως εκείνος επιθυμεί.

Συνοψίζοντας, το MessagePack προσφέρει μια συμπαγή, αποδοτική και γλωσσικά ανεξάρτητη μορφή serialization κατάλληλη για κρίσιμες σε απόδοση εφαρμογές. Η υποστήριξή του για διάφορους τύπους δεδομένων, το data streaming και η συμβατότητα μεταξύ πλατφορμών το καθιστούν μια ευέλικτη επιλογή για έργα που απαιτούν serialization και μετάδοση δεδομένων υψηλής απόδοσης, όπως για παράδειγμα τα βιντεοπαιχνίδια.

## 2.6 Foreign Function Interfaces

Στη ενότητα αυτή θα γίνει μία βασική ανάλυση των λειτουργιών των Foreign Function Interfaces (FFI) ξεκινώντας από τον ορισμό τους. Στη συνέχεια εξηγούνται τρία βασικά σημεία που χρίζουν προσοχής κατά τη δημιουργία και τον σχεδιασμό ενός FFI με αυτά να είναι το λεγόμενο Data Marshalling, η διαχείριση των σφαλμάτων και τέλος η υπολογιστική βαρύτητα που επιφέρει το κάλεσμα μεθόδων μεταξύ διαφορετικών γλωσσών.

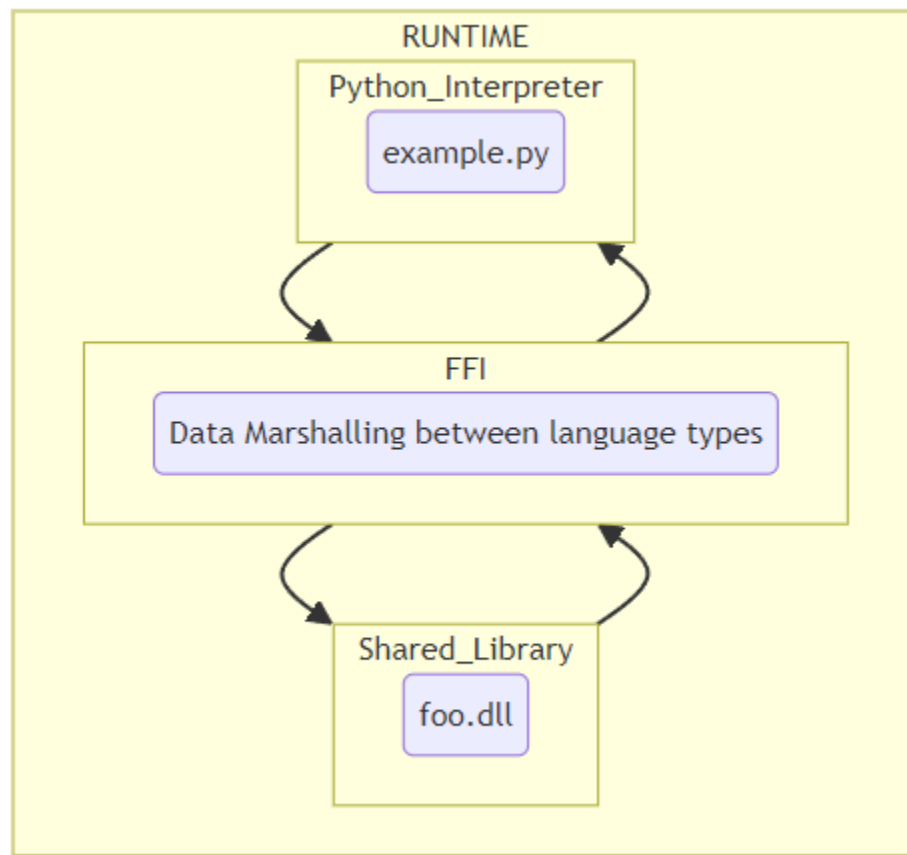
### 2.6.1 Ορισμός του Foreign Function Interface

Τα Foreign Function Interfaces (FFIs) αποτελούν έναν σημαντικό μηχανισμό που διευκολύνει τη δια λειτουργικότητα μεταξύ διαφορετικών γλωσσών προγραμματισμού, επιτρέποντας έτσι την απρόσκοπτη ενσωμάτωση διαφορετικών στοιχείων σε ένα γενικότερο περιβάλλον μίας εφαρμογής.

Στον τομέα ανάπτυξης λογισμικού, τα FFI χρησιμεύουν ως γέφυρες, επιτρέποντας στον κώδικα που είναι γραμμένος σε μια γλώσσα (host language) να έχει πρόσβαση σε συναρτήσεις και δομές δεδομένων που ορίζονται σε μια άλλη γλώσσα (guest language) ή και το ανάποδο. Αυτή η ικανότητα είναι καθοριστική σε σενάρια που απαιτούν τη χρήση πολλαπλών γλωσσών σε ένα ενιαίο project, όπως η αξιοποίηση των πλεονεκτημάτων απόδοσης των βιβλιοθηκών low-level γλωσσών όπως η C ή C++ από γλώσσες high-level όπως η C#, Python ή η JavaScript (ref). Η διαδικασία αυτή διακρίνεται στην Εικόνα 15.

Μέσω των FFI, οι προγραμματιστές μπορούν να ξεπεράσουν τα γλωσσικά εμπόδια, αξιοποιώντας τα πλεονεκτήματα των διαφόρων γλωσσών και εξασφαλίζοντας παράλληλα τη συμβατότητα και τη συνοχή σε ολόκληρη τη βάση του κώδικα. Ωστόσο, αυτή η αρχιτεκτονική λύση φέρει μαζί της και τα ανάλογα προβλήματα και εμπόδια, όπως είναι το data marshalling μεταξύ των δύο γλωσσών που θα πρέπει να διαχειριστεί ο εκάστοτε προγραμματιστής και το performance overhead που υπάρχει λόγω των εξωτερικών μεθόδων που καλούνται αλλά και λόγω του context switching που συμβαίνει εκείνη

τη στιγμή στον επεξεργαστή. Τέλος, ένας τομέας των FFI που χρήζει σημαντικής προσοχής είναι το error handling μεταξύ των γλωσσών (ref).



Εικόνα 15: FFI και Data Marshalling

### 2.6.2 Data Marshalling στα Foreign Function Interfaces

Η «διακίνηση δεδομένων» ή αλλιώς Data Marshalling, επίσης γνωστό ως “packing”, περιλαμβάνει τη μετατροπή δεδομένων από μια αναπαράσταση σε μια άλλη, συνήθως με σκοπό τη μεταφορά τους μεταξύ διαφορετικών συστημάτων ή γλωσσών. Στο πλαίσιο των FFI, το data marshalling συνήθως συνδέεται άμεσα με το serialization και deserialization και χρησιμοποιείται για τη μετατροπή δεδομένων μεταξύ των αναπαραστάσεων που χρησιμοποιούνται από την καλούσα γλώσσα και την καλούμενη γλώσσα, όπως αυτό φαίνεται στην Εικόνα 15. Για παράδειγμα εάν μια συνάρτηση σε μια βιβλιοθήκη C αναμένει μία συμβολοσειρά string και καλείται από την Python, η οποία συνήθως χρησιμοποιεί συμβολοσειρές Unicode, το FFI θα πρέπει να διαμορφώσει τη συμβολοσειρά Unicode της Python σε μια μορφή που μπορεί να κατανοήσει η συνάρτηση C, όπως ένα null-terminated ASCII string.

Μια παρόμοια διαδικασία μπορεί να εννοηθεί και μεταξύ των Εικόνων 16 και 17, όπου ο προγραμματιστής θα έπρεπε να είχε δημιουργήσει και τα δυο data types για να μπορέσει να «μεταφέρει» την πληροφορία και τα δεδομένα από την πλευρά της Python στη βιβλιοθήκη της C++ με το FFI να διαχειρίζεται τη σωστή «λήψη» και «αποστολή» των δεδομένων.

```

ffiSample.py > Data
1  # Imports the necessary C types
2  import ctypes
3
4  # Base class definition
5  class Data(ctypes.Structure):
6      _fields_ = [
7          ('intValue', ctypes.c_int),
8          ('floatValue', ctypes.c_float),
9          ('boolValue', ctypes.c_bool)
10 ]

```

Εικόνα 16: Data Container δείγμα σε Python προς μεταφορά σε C++ library

```

C++ ffiSample.cpp > ...
1  /* A simple C++ struct
2  with data type mirroring
3  of the Python Data class */
4  struct Data
5  {
6      int intValue;
7      float floatValue;
8      bool boolValue;
9  };

```

Εικόνα 17: Data Container δείγμα σε C++ έτοιμο να δεχτεί δεδομένα από μία κλάση Python μέσω του FFI

### 2.6.3 Διαχείριση των Exceptions μεταξύ των γλωσσών

Όταν καλείται μία συνάρτηση σε διαφορετικές γλώσσες, είναι σημαντικό να εξετάζεται ο τρόπος διάδοσης και χειρισμού των σφαλμάτων μεταξύ των διαφορετικών γλωσσών. Αυτό μπορεί να περιλαμβάνει τη μετατροπή των αναπαραστάσεων των σφαλμάτων μεταξύ των γλωσσών ή την παροχή μηχανισμών για τη μετάφραση των σφαλμάτων που ανακύπτουν σε μια γλώσσα σε exceptions ή κωδικούς σφαλμάτων σε μια άλλη γλώσσα (ref). Μερικοί ενδεικτικοί τρόποι σωστής διαχείρισης των σφαλμάτων μεταξύ διαφορετικών γλωσσών είναι οι ακόλουθοι:

- **Error Propagation:** Σε πολλές περιπτώσεις, τα σφάλματα που προκαλούνται από συναρτήσεις που καλούνται μέσω ενός FFI μεταδίδονται πίσω στον κώδικα που καλεί την συνάρτηση. Αυτό σημαίνει ότι εάν μια συνάρτηση που καλείται μέσω του FFI αντιμετωπίσει ένα σφάλμα, όπως ένα runtime

exception ή έναν κωδικό σφάλματος, μπορεί να κάνει throw ένα exception ή να επιστρέψει μια ένδειξη σφάλματος στον καλούντα κώδικα.

- **Κωδικοί Σφαλμάτων:** Τα FFI μπορούν να χρησιμοποιήσουν κωδικούς σφαλμάτων ή ειδικές τιμές επιστροφής για να υποδεικνύουν σφάλματα. Για παράδειγμα, μια συνάρτηση που καλείται μέσω ενός FFI μπορεί να επιστρέψει μια αρνητική τιμή ή έναν δείκτη NULL για να σηματοδοτήσει μια κατάσταση σφάλματος.

## 2.6.4 Performance Overhead στα Foreign Function Interfaces

Η χρήση των FFI συνεπάγεται μεταξύ άλλων και ζητήματα επιδόσεων, που αφορούν κυρίως τη μετατροπή δεδομένων, την επιβάρυνση κλήσης συναρτήσεων και τη διαχείριση μνήμης. Η μετατροπή δεδομένων μεταξύ διαφορετικών αναπαραστάσεων καθώς διασχίζει τα όρια των γλωσσών μπορεί να εισάγει μία υπολογιστική επιβάρυνση, ιδίως για μεγάλα σύνολα δεδομένων, καθώς συχνά περιλαμβάνει λειτουργίες marshalling και unmarshalling, όπως αυτές εξηγήθηκαν στην ενότητα 2.6.2. Παρομοίως, η πράξη της κλήσης συναρτήσεων μεταξύ διαφορετικών γλωσσών προσθέτει στο φορτίο επεξεργασίας λόγω βημάτων όπως η μεταφόρτωση παραμέτρων και η εναλλαγή περιβάλλοντος μεταξύ γλωσσών, το λεγόμενο context switch (ref). Επιπλέον, οι αποκλίσεις στους μηχανισμούς διαχείρισης μνήμης μεταξύ των γλωσσών μπορεί να επηρεάσουν την απόδοση, γεγονός που απαιτεί προσεκτική εξέταση, ιδίως σε σενάρια που περιλαμβάνουν συχνές λειτουργίες μνήμης όπως ένα library με cache και δομές δεδομένων.

## 2.7 Library Wrappers

Για να πλαισιωθεί σωστά μία πλήρης λειτουργική βιβλιοθήκη με σκοπό τη χρήση της από πολλαπλές γλώσσες, είναι ζωτικής σημασίας κομμάτι της ο λεγόμενος library wrapper, γνωστός και ως DLL Wrapper. Για την σωστή κατανόηση αυτού όμως χρειάζεται η διαλεύκανση κάποιων βασικών εννοιών όπως αυτές του managed κώδικα έναντι του unmanaged κώδικα, οι οποίες αποτελούν δύο από τους βασικούς πυλώνες γνώσης σχετικά με την διαχείριση μνήμης και δεδομένων στην Πληροφορική. Σε αυτή την ενότητα θα αναλυθεί η διαφορά του Managed κώδικα έναντι του Unmanaged κώδικα, θα δοθεί ένας βασικός ορισμός σχετικά με τους DLL wrappers και τέλος θα εξηγηθούν τα βασικά περιβάλλοντα χρήσης τους μέσω διαφορετικών τρόπων διεπαφών με εξωτερικό κώδικα.

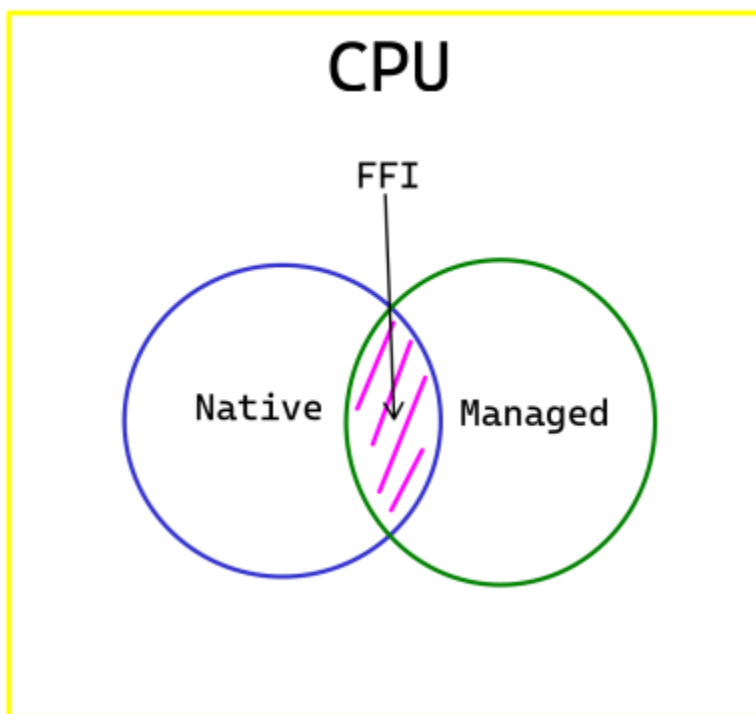
### 2.7.1 Managed και Unmanaged κώδικας

Όπως προαναφέρθηκε, ο managed και ο unmanaged κώδικας είναι θεμελιώδεις έννοιες στην ανάπτυξη λογισμικού, ιδίως στο πλαίσιο των γλωσσών και των περιβαλλόντων προγραμματισμού. Συγκεκριμένα, ο managed κώδικας λειτουργεί σε ένα διαχειριζόμενο περιβάλλον, το οποίο συνήθως πλαισιώνεται από ένα περιβάλλον runtime, όπως το Common Language Runtime (CLR) στο .NET Framework ή η Java Virtual Machine (JVM) στη Java. Αυτό το περιβάλλον παρέχει αυτόματη διαχείριση μνήμης, συμπεριλαμβανομένων χαρακτηριστικών όπως το Garbage Collection (GC), η οποία χειρίζεται τις εργασίες κατανομής και απομάκρυνσης μνήμης. Ο managed κώδικας είναι γνωστός για το υψηλότερο επίπεδο αφαίρεσης και την ανεξαρτησία πλατφόρμας του, καθιστώντας την ανάπτυξη,

το debugging και τη συντήρηση του ευκολότερη. Παραδείγματα γλωσσών που μεταγλωττίζονται σε managed κώδικα είναι η C#, η Visual Basic .NET, η Java και ορισμένες υλοποιήσεις της Python, όπως η Jython ή η IronPython όπου η τελευταία αποτελεί μία υλοποίηση της Python με bindings στη .NET (ref).

Αντίθετα, ο unmanaged, ή αλλιώς “native” κώδικας εκτελείται απευθείας στο hardware του συστήματος χωρίς την παρέμβαση ενός περιβάλλοντος runtime. Συνήθως γραμμένος σε γλώσσες όπως η C ή η C++, ο unmanaged κώδικας παρέχει στους προγραμματιστές άμεσο έλεγχο των εργασιών διαχείρισης μνήμης, συμπεριλαμβανομένης της κατανομής και της αποδέσμευσης της. Ενώ ο unmanaged κώδικας μπορεί να προσφέρει καλύτερες επιδόσεις και στενότερο έλεγχο των πόρων του συστήματος, απαιτεί προσεκτική διαχείριση της μνήμης και μπορεί να είναι ευάλωτος σε θέματα όπως memory leaks και ευπάθειες ασφαλείας, αν δεν αντιμετωπιστεί σωστά. Παραδείγματα μη διαχειριζόμενου κώδικα περιλαμβάνουν native κώδικα που έχει μεταγλωττιστεί από C ή C++, καθώς και κώδικα σε assembly (ref).

Όπως μπορεί να διακριθεί στην Εικόνα 18, υπάρχει δυνατότητα διεπαφής μεταξύ των δύο αυτών τύπων γλωσσών, μέσω του προ αναφερόμενου Foreign Function Interface (FFI) όπως αυτό αναλύθηκε στην ενότητα 2.6.



Εικόνα 18: Γεφύρωση managed και unmanaged κώδικα μέσω ενός FFI

Συνοψίζοντας, οι βασικές διαφορές των δύο αυτών εννοιών είναι η «ασφαλής», για τον managed κώδικα, διαχείριση της μνήμης σε αντίθεση με τον unmanaged κώδικα που δεν διαθέτει λειτουργίες όπως ο Garbage Collector για την αυτόματη αποδέσμευση αυτής και η αποδέσμευση της μνήμης πρέπει να γίνεται χειροκίνητα. Στη συνέχεια, ο unmanaged κώδικας υπερισχύει όσον αφορά την «ταχύτητα» του διότι δεν βασίζεται σε κάποιο runtime περιβάλλον, ενώ ο managed κώδικας παρέχει μία ανεξαρτησία σχετικά με την πλατφόρμα εκτέλεσής του και ευκολία ανάπτυξης.



Τέλος, ακριβώς λόγω αυτής της ανεξαρτησίας του σχετικά με την πλατφόρμα εκτέλεσης του, ο managed κώδικας είναι πιο φορητός σε διαφορετικές πλατφόρμες, ενώ ο unmanaged κώδικας μπορεί να χρειαστεί να ξανά γίνει compile εκ νέου ή να προσαρμοστεί για διαφορετικές πλατφόρμες.

## 2.7.2 Ορισμός των Library/DLL Wrappers στην Πληροφορική

Οι DLL wrappers, γνωστοί και ως dynamic-link library wrappers, είναι components λογισμικού ή βιβλιοθήκες που έχουν σχεδιαστεί για να απλοποιούν τη χρήση των dynamically-linked libraries (DLL) σε ένα συγκεκριμένο περιβάλλον προγραμματισμού. Παρέχουν έναν τρόπο για τις εφαρμογές να τμηματοποιήσουν τον κώδικα τους σε λογικές βιβλιοθήκες και να μοιραστούν πόρους (ref), όπως για παράδειγμα διακρίνεται στην Εικόνα 15.

Οι wrappers αυτοί κατά επέκταση, προσφέρουν ένα επίπεδο αφαίρεσης ή higher-level διεπαφής γύρω από τις βιβλιοθήκες DLL, καθιστώντας ευκολότερη τη χρήση τους εντός μιας συγκεκριμένης γλώσσας προγραμματισμού ή ενός συγκεκριμένου πλαισίου. Συχνά παρέχουν μια πιο φιλική προς το προγραμματιστή διεπαφή, αποκρύπτοντας τις πολυπλοκότητες της άμεσης αλληλεπίδρασης με τα DLL αυτά (ref).

Συγκεκριμένα, μέσω των wrappers μπορεί να επιτευχθεί η συμβατότητα με ποικίλες γλώσσες, για παράδειγμα μία διεπαφή ενός C++ DLL με την C# της .NET ή ακόμα και με ένα περιβάλλον Python και επιτυγχάνεται η απλοποίηση του API (Application Programming Interface) ενός σύνθετου DLL με τη χρήση higher-level γραφής και αρχιτεκτονικής. Επιπλέον, μέσα σε έναν wrapper μπορούν να ενσωματωθούν οι προαναφερθέντες μηχανισμοί exception handling (εν. 2.6.3) για τη διαχείριση των exceptions ή των κωδικών σφαλμάτων που μπορεί να επιστρέψει μία συνάρτηση από το DLL. Τέλος, σε έναν τέτοιου τύπου wrapper μπορούν να υλοποιηθούν και μηχανισμοί σωστής δέσμευσης και αποδέσμευσης της μνήμης για την ομαλή λειτουργία της εκάστοτε εφαρμογής (ref).

## 2.7.3 Managed και Unmanaged code wrappers

Οι managed και unmanaged code wrappers ακολουθούν τη θεωρία που παρουσιάστηκε στην ενότητα 2.7.1. Χρησιμοποιούνται και οι δύο για να γεφυρώσουν το χάσμα μεταξύ managed και unmanaged περιβαλλόντων κώδικα σε συνεργασία με τις λειτουργίες των FFIs, αλλά εξυπηρετούν διαφορετικούς σκοπούς και λειτουργούν με διαφορετικούς τρόπους. Οι managed wrappers χρησιμοποιούνται για να παρέχουν στον managed κώδικα (όπως η C#, η VB.NET, η Java ή άλλες γλώσσες) πρόσβαση σε unmanaged κώδικα, συνήθως DLL ή COM objects (ref about COM Objects). Χρησιμοποιούνται συχνά για την χρήση παλαιότερου κώδικα ή εξωτερικών βιβλιοθηκών που είναι γραμμένες σε γλώσσες όπως η C ή η C++ που δεν είναι άμεσα συμβατές με το .NET Framework. Αντίστοιχα, οι unmanaged wrappers, επίσης γνωστοί ως platform invoke (P/Invoke) wrappers, χρησιμοποιούνται για την κλήση managed κώδικα από unmanaged κώδικα, χωρίς όμως αυτό να περιορίζει τις Platform Invoke λειτουργίες μόνο σε περιβάλλοντα unmanaged κώδικα. Επιτρέπουν σε unmanaged κώδικα γραμμένο σε γλώσσες όπως η C ή η C++ να καλεί συναρτήσεις που ορίζονται σε managed σύνολα, συνήθως DLL που δημιουργούνται με γλώσσες .NET.

## 2.8 Συστήματα και λύσεις serialization στις μηχανές βιντεοπαιχνιδιών

Στη τελευταία αυτή ενότητα του θεωρητικού πλαισίου θα γίνει μία τυπική αναφορά στους μηχανισμούς serialization ή ακόμα και ολόκληρες λύσεις που διαθέτουν ενσωματωμένες δύο από τις διασημότερες μηχανές δημιουργίας βιντεοπαιχνιδιών, με αυτές να είναι οι Unity και Unreal Engine.

Στο επόμενο κεφάλαιο της Μεθοδολογίας (Κεφ. 3), παρουσιάζονται τα χαρακτηριστικά και ο σχεδιασμός του μηχανισμού που θα αναπτυχθεί με βάση όλη την προαναφερόμενη θεωρία.

### 2.8.1 Δυνατότητες serialization στη Unity

Η Unity προσφέρει διάφορους μηχανισμούς για το serialization δεδομένων και τη δημιουργία μηχανισμών αποθήκευσης στα project της, καλύπτοντας διαφορετικές ανάγκες και πολυπλοκότητες.

Στην απλούστερη μορφή του, οι προγραμματιστές μπορούν να χρησιμοποιήσουν το PlayerPrefs (ref from docs), ένα σύστημα αποθήκευσης τιμών-κλειδιών κατάλληλο για την αποθήκευση μικρών ποσοτήτων δεδομένων, όπως οι προτιμήσεις των παικτών και οι βασικές ρυθμίσεις του παιχνιδιού που καταγράφονται στο registry των συστημάτων. Σημειώνοντας προσαρμοσμένες κλάσεις με το attribute [Serializable], οι προγραμματιστές μπορούν κάνουν serialize και να deserialize την πολύπλοκη κατάσταση του παιχνιδιού, την πρόοδο του παίκτη και άλλα βασικά δεδομένα του παιχνιδιού. Αυτό φυσικά είναι εφικτό μόνο με τη δημιουργία ενός μηχανισμού αποθήκευσης διότι η Unity δεν παρέχει κάποιον τέτοιο μηχανισμό έτοιμο.

Τέλος, η Unity υποστηρίζει serialization σε μορφές JSON και XML μέσω των κλάσεων JsonUtility και XmlSerializer αντίστοιχα, παρέχοντας ευελιξία στις μορφές αποθήκευσης και ανταλλαγής δεδομένων (ref). Στη συνέχεια, για πιο αποδοτικό μέγεθος αρχείου και χειρισμό δεδομένων, η Unity υποστηρίζει binary serialization μέσω της κλάσης BinaryFormatter. Αυτή η μέθοδος ωστόσο δεν συστήνεται διότι με βάση το επίσημο documentation της .NET η κλάση BinaryFormatter έγινε Obsolete (ref) στην έκδοση .NET 8.

Αυτές οι επιλογές serialization δίνουν τη δυνατότητα στους προγραμματιστές να δημιουργήσουν συστήματα αποθήκευσης προσαρμοσμένα στις απαιτήσεις του παιχνιδιού τους.

### 2.8.2 Δυνατότητες serialization στην Unreal Engine

Η Unreal Engine προσφέρει μια μεγάλη ποικιλία εργαλείων για το serialization δεδομένων και τη δημιουργία συστημάτων αποθήκευσης. Πρώτον, οι προγραμματιστές μπορούν να αξιοποιήσουν την κλάση SaveGameObject για να ορίσουν προσαρμοσμένες δομές δεδομένων για την αποθήκευση της προόδου του παιχνιδιού. Αυτή η κλάση διευκολύνει το serialization των δεδομένων του παιχνιδιού και μπορεί να γίνει ακόμα και subclassed μέσω της κλάσης USaveGame έτσι ώστε να υλοποιηθούν οι απαραίτητες μέθοδοι serialization ανάλογα με τις απαιτήσεις του παιχνιδιού (ref).

Επιπλέον, η Unreal Engine παρέχει ένα σύνολο συναρτήσεων, όπως οι SaveGameToSlot() και LoadGameFromSlot(), για τη διαδικασία αποθήκευσης και φόρτωσης δεδομένων από και προς το δίσκο σε συνεργασία με τις κλάσεις SaveGameObject. Η μηχανή υποστηρίζει επίσης data archiving και data compression, επιτρέποντας την αποτελεσματική αποθήκευση της κατάστασης των δεδομένων.

Τέλος, οι προγραμματιστές μπορούν να κάνουν χρήση του ενσωματωμένου συστήματος SaveGame, το οποίο αυτοματοποιεί μεγάλο μέρος της διαδικασίας αποθήκευσης και φόρτωσης, απλοποιώντας την υλοποίηση ενός συστήματος game saving στα project τους.

## Κεφάλαιο 3: Μεθοδολογία

### 3.1 Εισαγωγή κεφαλαίου

Στο κεφάλαιο αυτό θα γίνει μία ενδελεχής ανάλυση των απαιτήσεων και των χαρακτηριστικών που θα πρέπει να έχει η τελική λύση του πρακτικού μέρους της πτυχιακής αυτής και επιπλέον θα επιλεγεί ο βασικός εσωτερικός serializer της βιβλιοθήκης αλλά και η γλώσσα γραφής της.

Στη συνέχεια, θα σχεδιαστεί ο τρόπος με τον οποίο θα διατηρηθούν τα references μεταξύ των objects κατά το serialization, θα παρουσιαστεί ο αναλυτικός σχεδιασμός της βιβλιοθήκης, του Two-Fold FFI, του C# wrapper και της προτεινόμενης αρχιτεκτονικής χρήσης που θα τα περικλείει στο test environment.

Τέλος, γίνεται αναφορά στον βαθμό πραγματοποίησης των στόχων της διατριβής αλλά γίνεται και η ανάδειξη των σχεδιαστικών στοιχείων του πρακτικού μέρους.

### 3.2 Ανάλυση απαιτήσεων της συνολικής λύσης

Το open-source solution που πρόκειται να παρουσιαστεί μετέπειτα έχει σαν πυρήνα του μερικές βασικές αρχές όπου θα βασιστεί ο σχεδιασμός του.

Αρχικά, βασικά θεμιτά χαρακτηριστικά της λύσης αυτής είναι, όπως προαναφέρθηκε, η open source φύση του και με βάση αυτό πρέπει να ληφθούν υπόψη μερικά σημαντικά σημεία, όπως είναι το ποσοστό το οποίο ο εκάστοτε προγραμματιστής θα μπορεί να διευρύνει τις λειτουργίες της βιβλιοθήκης για να την σμιλέψει στα δικά του project. Έτσι καταλήγουμε σε μία βιβλιοθήκη που θα πρέπει να είναι αρκετά modular, versatile αλλά και extensible χωρίς όμως να χάνει την αρχική της δομή και λειτουργία. Ο κώδικας της θα βασιστεί με μία αρχιτεκτονική Component-driven (ref) έτσι ώστε να μπορούν πολλά σημεία της να αντικατασταθούν ή να επεκταθούν.

Στη συνέχεια, ένα επιπλέον βασικό χαρακτηριστικό της βιβλιοθήκης είναι η συμβατότητα της με διαφορετικές μηχανές βιντεοπαιχνιδιών, για τις οποίες και προβλέπεται η χρήση της, χωρίς ωστόσο αυτό να την περιορίζει. Για τον λόγο αυτό, θα πρέπει η βιβλιοθήκη και τα επί μέρους κομμάτια της να είναι language-agnostic, να μην περιορίζονται δηλαδή από την εξωτερική γλώσσα που θα τα περικλείει.

Τέλος, λόγο του ίδιου χαρακτηριστικού θα πρέπει να χρησιμοποιηθούν οι κατάλληλες δομές, τεχνικές και τρόποι γραφής, ώστε σαν σύνολο να μην έχει κάποιο μεγάλο αποτύπωμα μνήμης, κάτι που ως αποτέλεσμα θα επιφέρει και την βέλτιστη απόδοση. Φυσικά πάντα μέσα στα πλαίσια τα οποία επιτρέπουν οι δομές και οι τεχνικές που θα χρησιμοποιηθούν.

Τέλος, στόχος είναι η ελευθερία του serializer που θα χρησιμοποιεί ο εκάστοτε χρήστης, έτσι ώστε να μην αναγκαστεί να βασιστεί στον εσωτερικό τρόπο serialization που θα επιλεγεί στις επόμενες ενότητες με βασικό γνώμονα το human readability του και την απόδοση του ώστε να μην υπάρχει μεγαλύτερο αποτύπωμα μνήμης.

### 3.3 Ανάλυση επιλογής C++ ως βασική γλώσσα συγγραφής

Η επιλογή της χρήσης της C++ για ένα DLL εξαρτάται από διάφορους παράγοντες, όπως οι απαιτήσεις απόδοσης, η υπάρχουσα βάση κώδικα, η συμβατότητα με άλλες γλώσσες και η εξοικείωση

του προγραμματιστή. Η χρήση της C++ για μια βιβλιοθήκη DLL που προορίζεται για χρήση σε μηχανές παιχνιδιών προσφέρει πολλά πλεονεκτήματα.

Αρχικά, όπως προαναφέρθηκε στο κεφάλαιο του θεωρητικού πλαισίου, η C++ φημίζεται για τις εξαιρετικές επιδόσεις της και τον low-level έλεγχο πόρων και μνήμης που παρέχει, καθιστώντας την ιδανική επιλογή για την ανάπτυξη παιχνιδιών όπου η απόδοση είναι ζωτικής σημασίας (ref). Επιπλέον, πολλές μηχανές παιχνιδιών παρέχουν ισχυρή υποστήριξη για plugins και βιβλιοθήκες C++, διευκολύνοντας την ενσωμάτωση της παρούσας βιβλιοθήκης στη διαδικασία παραγωγής τους.

Επιπλέον, η φορητότητα του κώδικα που είναι γραμμένος σε C++ επιτρέπει το compile του για πολλαπλές πλατφόρμες, απλοποιώντας την ανάπτυξη παιχνιδιών πολλαπλών πλατφορμών. Αυτή η ευελιξία είναι ιδιαίτερα πολύτιμη στο σημερινό τοπίο των παιχνιδιών, όπου τα παιχνίδια αναμένεται συχνά να τρέχουν σε διάφορες συσκευές και λειτουργικά συστήματα.

Τέλος, η εκτεταμένη διαθεσιμότητα βιβλιοθηκών και frameworks που σχετίζονται με παιχνίδια και είναι γραμμένα σε C++ απλοποιεί τις ανάπτυξής τους, παρέχοντας στους προγραμματιστές πληθώρα πόρων και εργαλείων και καθιστώντας την παρούσα βιβλιοθήκη συμβατή με πολλά από αυτά τα plugins (ref).

Σχετικά με το υπόλοιπο σκέλος της λύσης, η επιλογή της C# για τη γλώσσα που θα πλαισιώσει το wrapper αποτελεί μονόδρομο διότι η Unity χρησιμοποιεί αποκλειστικά C# σα βασική γλώσσα συγγραφής (ref). Ωστόσο, αυτό θα βοηθήσει και στην παρουσίαση των αποτελεσμάτων της λύσης μέσω χρήσης της για την λειτουργία την οποία προορίζεται.

### 3.4 Επιλογή βασικού serializer

Τόσο το MessagePack όσο και οι Protocol Buffers (Protobuf) είναι μορφότυποι serialization που χρησιμοποιούνται για την αποτελεσματική κωδικοποίηση (encoding) και αποκωδικοποίηση (decoding) δομημένων δεδομένων. Ωστόσο, έχουν διαφορετικές φιλοσοφίες σχεδιασμού και εφαρμογές, που επηρεάζουν την επιλογή μεταξύ τους, όπως αυτές αναλύθηκαν στην ενότητα 2.5.

Πρώτον, το MessagePack χαρακτηρίζεται για την απλότητα και την ευκολία χρήσης του, παρέχοντας μια απλή μορφή που βοηθά στο debugging και την κατανόηση των δεδομένων, λόγω της μετάφρασης των binary δεδομένων του σε μορφή JSON. Η ανώτερη απόδοσή του, ιδίως όσον αφορά την ταχύτητα κωδικοποίησης και αποκωδικοποίησης, πλεονεκτεί για απαιτητικές εφαρμογές, όπως και αυτό διακρίνεται στην Εικόνα 14. Επιπλέον, η υποστήριξη του MessagePack σε διάφορες γλώσσες προγραμματισμού το καθιστά πολύ καλή επιλογή για ενσωμάτωση σε ετερογενή περιβάλλοντα, ενισχύοντας την ευελιξία του.

Ένα άλλο αξιοσημείωτο πλεονέκτημα του MessagePack είναι η runtime δυνατότητα του να κάνει serialize κάθε είδος τύπου, που του επιτρέπει να χειρίζεται ποικίλους τύπους δεδομένων χωρίς προκαθορισμένα schemes. Αυτή η ευελιξία είναι ιδιαίτερα πολύτιμη σε σενάρια με δυναμικές απαιτήσεις δεδομένων, προσφέροντας προσαρμοστικότητα χωρίς να θυσιάζεται η αποδοτικότητα.

Από την άλλη πλευρά, τα protobufs προσφέρουν διακριτά πλεονεκτήματα που μπορεί να είναι προτιμότερα σε ορισμένα πλαίσια. Με τα protobufs, η επιβολή κάποιου schema εξασφαλίζει ισχυρές εγγυήσεις δομής δεδομένων κατά το serialization και το deserialization, προωθώντας την ακεραιότητα των δεδομένων. Επιπλέον, η λειτουργία code generation παράγει κλάσεις που αφορούν τη γλώσσα με βάση το schema της, απλοποιώντας την ενσωμάτωση σε βάσεις κώδικα της C++ και παρέχοντας type-safety. Επιπλέον, τα protobufs έχουν σχεδιαστεί με γνώμονα το backwards compatibility, επιτρέποντας

το extension των ήδη υπάρχοντων schemas δεδομένων χωρίς να διαταράσσεται η συμβατότητα με τα υπάρχοντα serialized δεδομένα.

Συμπερασματικά, η επιλογή μεταξύ MessagePack και protobufs εξαρτάται από συγκεκριμένες απαιτήσεις και περιορισμούς. Το MessagePack ωστόσο υπερέχει ως προς την απλότητα, τις επιδόσεις και τη γλωσσική ανεξαρτησία, ενώ τα protobufs προτιμώνται για την επιβολή schema, τις δυνατότητες δημιουργίας κώδικα και το backwards compatibility. Ωστόσο, αξίζει να σημειωθεί ότι η διαλειτουργικότητα του MessagePack με το JSON ενισχύει περαιτέρω τη χρησιμότητά του, επιτρέποντας την ομαλή μετάφραση σε μορφή JSON και διευκολύνοντας την επικοινωνία με συστήματα που βασίζονται στο JSON για την ανταλλαγή δεδομένων. Τέλος, είναι και open source που αυτό επιτρέπει σε κάθε προγραμματιστή να το σμιλέψει στα χαρακτηριστικά που αυτός επιθυμεί.

Με βάση αυτές τις πληροφορίες όπως αναφέρθηκαν επιγραμματικά σε αυτή την ενότητα αλλά και στην ενότητα 2.5 όπως επίσης και με γνώμονα τους βασικούς στόχους περί επιλογής serializer στη παρούσα βιβλιοθήκη, θα χρησιμοποιηθεί το C++ MsgPack (ref) για τη διαδικασία του serialization.

### 3.5 Χαρακτηριστικά λύσης

Σε αυτή την ενότητα θα αναλυθούν επιγραμματικά τα χαρακτηριστικά που θα πρέπει να έχουν η C++ βιβλιοθήκη DLL, το ενδιάμεσο FFI και ο C# wrapper.

#### 3.5.1 Χαρακτηριστικά C++ DLL

Βασικά χαρακτηριστικά της C++ βιβλιοθήκης αποτελούν η διαχείριση των I/O λειτουργιών, το caching των δεδομένων προς serialization και η διαχείριση των SMRI (βλ. εν. 3.6). Επιπλέον, θα δύναται στο χρήστη πρόσβαση στα δεδομένα προς serialization, το update αυτών αλλά και η διαγραφή τους μέσω ενός εκλεπτυσμένου API. Τέλος, σε συνεργασία με το FFI σε περίπτωση κάποιου exception θα επιστρέφονται πίσω error codes, όπου αυτό είναι εφικτό.

#### 3.5.2 Χαρακτηριστικά του Foreign Function Interface

Αρχικά το Foreign Function Interface (FFI) θα αποτελείται από δύο μέρη, την πλευρά και τη διαχείριση των δομών, της μνήμης και άλλων, μέσα στη βιβλιοθήκη της C++ αλλά και τη διαχείριση των δεδομένων κατά τη μεταφορά τους για serialization στη πλευρά της C# στη Unity. Φυσικά, και στις δύο πλευρές θα υπάρχει μηχανισμός error handling για τους κωδικούς και τα exceptions του library όπου επεκτείνεται στον C# wrapper.

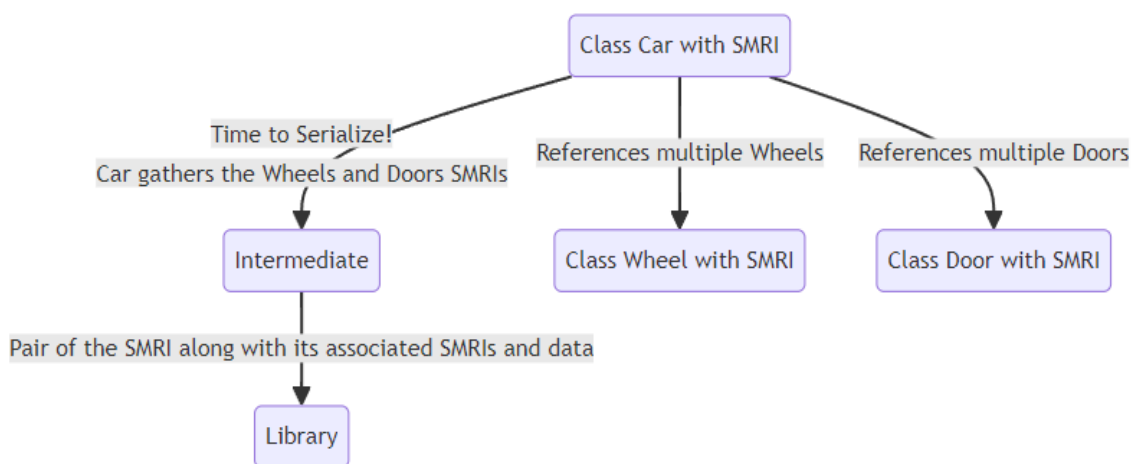
#### 3.5.3 Χαρακτηριστικά του C# wrapper

Κλείνοντας την ενότητα των χαρακτηριστικών, ο C# wrapper στην πλευρά της Unity θα κάνει encapsulate τις wrapped μεθόδους μέσα από το C++ DLL, θα έχει ένα ευανάγνωστο API με στόχο την πιο εύκολη πρόσβαση από τον προγραμματιστή. Τέλος, σε συνεργασία με το FFI θα παρέχει την κατάλληλη διαχείριση των error codes, των exceptions και θα αναλαμβάνει και τη μεταφορά των δεδομένων στη βιβλιοθήκη για αποθήκευση.

### 3.6 Διατήρηση των references μετά το serialization

Όπως αναλύθηκε στην ενότητα 2.4.3, δεν υπάρχει κάποιος απτός τρόπος να συντηρηθούν τα references μεταξύ των instances κατά το serialization. Για αυτό τον λόγο, ένας μηχανισμός συντήρησης αυτών θα χρειαστεί να δημιουργηθεί. Μία πάγια τακτική είναι η συσχέτιση ενός value με ένα ακέραιο αριθμό (int).

Ο λεγόμενος SMRI (Snapshot Manager Reference Index) θα είναι υπεύθυνος να μεταφέρεται μαζί με τις πληροφορίες πίσω στη βιβλιοθήκη και θα γίνεται και αυτός μαζί serialize, έτσι ώστε μετά το deserialization να μπορεί να ξαναγίνει η συσχέτιση των δεδομένων μεταξύ τους, όπως αυτό διακρίνεται στην Εικόνα 19.



Εικόνα 19: Σχεδιάγραμμα λογικής μηχανισμού SMRI

Ο παραπάνω μηχανισμός θα αναλυθεί εις βάθος στην ενότητα 3.8.

### 3.7 Περιβάλλον δοκιμών στη Unity

Σχετικά με τις δοκιμές, θα στηθεί ένα απλό περιβάλλον με primitive Game Objects (ref) με μερικά custom χαρακτηριστικά όπου θα γίνει χρήση του πλήρους μηχανισμού που θα σχεδιαστεί στη συνέχεια. Τα δεδομένα θα αντικατοπτρίζουν ένα πραγματικό σενάριο παιχνιδιού ώστε να υπάρχει μία σωστή διαδικασία debugging του library αλλά και του serialization.

### 3.8 Σχεδιασμός λύσης

Παρακάτω ακολουθεί ο αναλυτικός θεωρητικός σχεδιασμός του τεχνικού μέρους της πτυχιακής εργασίας.

#### 3.8.1 Σχεδιασμός βασικής βιβλιοθήκης σε C/C++

Η βιβλιοθήκη θα διαχειρίζεται το λεγόμενο current SMRI, παρέχοντας μεθόδους για την αύξηση, μείωση αλλά και επαναφορά αυτού πίσω στην προκαθορισμένη του αρχική τιμή του -1. Είναι απαραίτητη η σωστή διαχείριση αυτού, διότι μέσω του SMRI θα γίνεται και ολόκληρη η διαφοροποίηση των δεδομένων σε πακέτα.

Στη συνέχεια, η βιβλιοθήκη θα παρέχει μεθόδους για τη διαχείριση I/O και συγκεκριμένα θα έχει accessor και mutator μεθόδους για το save directory μέσω absolute path τύπου string αλλά και αντίστοιχα θα έχει mutator και accessor μεθόδους για το όνομα του παραχθέντος αρχείου από το οποίο θα κληθεί στην περίπτωση του deserialization, να διαβάσει τα serialized δεδομένα ώστε να ξανά επαναφέρει τα δεδομένα στην προηγούμενη τους μορφή.

Επιπλέον, η βιβλιοθήκη θα δέχεται τα δεδομένα σε μορφή πινάκων(arrays) τύπου char που θα αποθηκεύονται στη βιβλιοθήκη και θα περνάνε απευθείας μία διαδικασία containerization, δηλαδή αντί να αποθηκεύονται απευθείας μέσα σε μία δομή αποθήκευσης θα αντιγράφονται by-value μέσα σε ειδικά διαμορφωμένα struct instances και μετά αυτά τα instances θα αποθηκεύονται σε μία δομή δεδομένων. Αυτά τα instances, θα είναι προσβάσιμα μέσω του SMRI τους για διαγραφή από τη δομή, την επιστροφή των δεδομένων τους πίσω στην host γλώσσα αλλά και τη λήψη των Referenced SMRI τους. Για την αποτελεσματική μεταφορά των δεδομένων θα προτιμηθεί η μεταφορά ενός pointer στην αρχή του πίνακα (array) των δεδομένων προς αποθήκευση και όχι ένας συγκεκριμένος τύπος δεδομένων από τη host γλώσσα προς τη βιβλιοθήκη. Αυτό καθιστά γρήγορη και αποτελεσματική την διαχείριση των δεδομένων εσωτερικά της βιβλιοθήκης και επίσης δεν αναγκάζει τον εκάστοτε προγραμματιστή να δημιουργεί διπλότυπα structs και instances των δεδομένων που θέλει να αποθηκεύσει και στην host γλώσσα αλλά και μέσα στην βιβλιοθήκη π.χ. ένα C# struct τύπου “MyData” που θα έπρεπε να δημιουργηθεί και στην πλευρά της C# αλλά και στην πλευρά της C++ ώστε να γίνει το απαραίτητο data marshalling κατά τη μεταφορά.

Επιπροσθέτως, θα υπάρχουν οι απαραίτητες μέθοδοι για packing(serialization) και unpacking(deserialization) των δεδομένων από το προαναφερθείς save directory αλλά και μέθοδοι για την manual εκκαθάριση της μνήμης της βιβλιοθήκης και επαναφοράς των τιμών της.

Τέλος, ολόκληρη η διαχείριση των σφαλμάτων της βιβλιοθήκης θα γίνεται μέσω error codes που θα δίνονται μαζί με την βιβλιοθήκη.

### 3.8.2 Σχεδιασμός του Two-Fold Foreign Function Interface

Όπως αναφέρθηκε στο κεφ. 2.6, ένα FFI είναι υπεύθυνο για τη σωστή διεκπεραίωση του data marshalling και τη μετάδοση των σφαλμάτων ή των κωδικών αυτών, στην host γλώσσα, οποιαδήποτε κι αν είναι αυτή. Για να μπορέσει αυτό να λειτουργήσει αποτελεσματικά στην παρούσα βιβλιοθήκη το FFI θα αποτελείται από δύο μέρη, την πλευρά της βιβλιοθήκης από όπου θα προετοιμάζονται τα δεδομένα των μεθόδων ή οι κωδικοί σφάλματος για επιστροφή στην host γλώσσα μετά το πέρας της λειτουργίας τους και τη δεύτερη πλευρά όπου θα ενσωματωθεί μέσα στον C# wrapper ώστε αυτός με τη σειρά του να κάνει το απαραίτητο marshalling των τύπων και των δεδομένων για αποστολή ή λήψη από τη βιβλιοθήκη. Ακριβές παράδειγμα αυτής της διαδικασίας, διακρίνεται στην Εικόνα 18.

### 3.8.3 Σχεδιασμός C# Wrapper

Όπως αναλύθηκε στο κεφ. 2.7, ένας language wrapper είναι υπεύθυνος για το encapsulation των μεθόδων μίας γλώσσας για χρήση σε μία άλλη γλώσσα, που στο περιβάλλον της παρούσας πτυχακής



εργασίας η host γλώσσα είναι η C# και η guest γλώσσα είναι η C++ λόγω της φύσης του DLL. Με βάση τα χαρακτηριστικά του κεφ. 3.5.3 λοιπόν, ο static C# wrapper θα παρέχει μεθόδους όπου εμφανισιακά θα είναι πανομοιότυπες με τις μεθόδους της βιβλιοθήκης αλλά θα είναι ειδικά διαμορφωμένες για το managed περιβάλλον της C#. Συγκεκριμένα, μέσω της χρήσης των Platform Invocation Services του .NET θα γίνονται οι απαραίτητες κλήσεις στις μεθόδους της βιβλιοθήκης αλλά και η διαχείριση των επιστρεφόμενων error codes για να μπορεί ο εκάστοτε προγραμματιστής να κάνει το απαραίτητο debugging σε περίπτωση κάποιου σφάλματος.

### 3.9 Σχεδιασμός test environment στη μηχανή Unity

Στο test environment θα γίνει μία προσομοίωση ενός πραγματικού σεναρίου παιχνιδιού το οποίο θα περιλαμβάνει δεδομένα τύπου double, float, string, int, Vector3 και Quaternion. Φυσικά, για να μπορέσει να ενσωματωθεί σωστά η βιβλιοθήκη μέσα σε ένα παιχνίδι αλλά και για να φανεί η χρήση των SMRIs, στο ακόλουθο κεφάλαιο θα αναλυθεί και σχεδιαστεί η αρχιτεκτονική χρήσης της βιβλιοθήκης σε ένα περιβάλλον αντικειμενοστραφές προγραμματισμού. Αυτό φυσικά δεν περιορίζει την χρήση της βιβλιοθήκης εκτός παιχνιδιών.

#### 3.9.1 Δεδομένα και αντικείμενα

Το test environment θα προσομοιώνει ένα shooter περιβάλλον στο οποίο ο εκάστοτε παίχτης έχει στην κατοχή του ένα inventory, το οποίο inventory με τη σειρά του περιέχει έναν N αριθμό όπλων.

Τα δεδομένα που θα πρέπει να αποθηκευτούν είναι τα ακόλουθα:

1. Player
  - a. uint Smri
  - b. int[] RefSmris
  - c. float \_Health
  - d. float \_Stamina
  - e. float \_Shield
  - f. bool \_IsAlive
  - g. Vector3 \_Position
  - h. Quaternion \_Rotation
2. Inventory
  - a. uint Smri
  - b. int[] RefSmris
  - c. int \_MaxItems
  - d. Vector3 \_Position
  - e. Quaternion \_Rotation
3. Weapon
  - a. uint Smri
  - b. int[] RefSmris

- c. `int _Ammo`
- d. `bool _Loaded`
- e. `Vector3 _Position`
- f. `Quaternion _Rotation`

Τα παραπάνω δεδομένα είναι σχεδιασμένα να αναδείξουν τις ικανότητες της βιβλιοθήκης στον χειρισμό των δεδομένων εσωτερικά αυτής αλλά και την ορθή χρήση των SMRIs σε ένα development περιβάλλον.

### 3.9.2 Αρχιτεκτονική χρήσης SMRI

Τα Snapshot Manager Reference Indexes (SMRIs) αποτελεί το κύριο χαρακτηριστικό διαχείρισης των δεδομένων εσωτερικά και κατά συνέπεια εξωτερικά της βιβλιοθήκης. Δρουν, σαν *unique identifiers* για τα πακέτα δεδομένων και στην πραγματικότητα η βιβλιοθήκη θα είναι υπεύθυνη για την σωστή καταχώριση των πακέτων αυτών μέσα στις δομές αποθήκευσης της. Για να επιτευχθεί αυτό, σε ένα αντικειμενοστραφές περιβάλλον, θα σχεδιαστεί και θα προταθεί μία «αρχιτεκτονική χρήσης» ώστε να μπορέσει η βιβλιοθήκη να αξιοποιηθεί στο μέγιστο της.

Συγκεκριμένα, στο loading δεδομένων στα βίντεο παιχνίδια υπάρχουν δύο βασικά σενάρια loading. Το σενάριο στο οποίο ο κόσμος αποθηκεύεται ολόκληρος με τη χρήση κάποιων metadata και μετέπειτα φορτώνεται πίσω στο ίδιο στάδιο και μορφή που ήταν κατά τη διάρκεια του save και το σενάριο στο οποίο ο κόσμος φορτώνεται σε ένα default state – σαν δηλαδή να φορτώνεται η σκηνή και τα δεδομένα όπως θα φορτωνόταν σε ένα καινούργιο save – και μετέπειτα ένας μηχανισμός περνάει όλα τα αποθηκευμένα δεδομένα από το αρχείο μέσα στο παιχνίδι σε μία σειριακή αλληλουχία με συγκεκριμένη σειρά. Η προτεινόμενη αρχιτεκτονική θα βασιστεί στο δεύτερο σενάριο.

Οι κλάσης και τα αντικείμενα που ακολουθούν θα ανήκουν στο namespace Proposed Architecture:

1. Common
  - a. Θα αποτελεί το μόνο Singleton(ref) του παιχνιδιού και θα είναι υπεύθυνη για το instance caching των World Loader και Save Manager.
2. Global Properties
  - a. Θα περιέχει constants όπως το save folder path και το save folder name για χρήση από οποιοδήποτε σημείο του παιχνιδιού.
3. World Loader
  - a. Θα είναι υπεύθυνη για το σωστό φόρτωμα των αντικειμένων του Player, Inventory και Weapon ανάλογα με το load state του παιχνιδιού, εάν αυτό θα είναι “Fresh Save” ή “From Load”. Στην περίπτωση που το παιχνίδι ξεκινάει σε “From Load” state, ο World Loader θα είναι υπεύθυνος να καλέσει και την ανάλογη μέθοδο από τον -ακόλουθο- Save Manager μέσω του Common singleton.
4. Save Manager
  - a. Η κλάση Save Manager θα είναι υπεύθυνη να κάνει encapsulate τον C# wrapper ο οποίος περιέχει μέσα τις static μεθόδους που με την σειρά τους καλούν τις μεθόδους από την βιβλιοθήκη. Συγκεκριμένα, θα περιέχει μεθόδους για το registering των

ISnapshots -αναλύονται αμέσως μετά- και των ISnapshot Models -αναλύονται αμέσως μετά- όπως και για το unregistering τους. Επιπλέον, θα περιέχει μεθόδους για το “Saving” και το “Loading” των δεδομένων μέσα από την βιβλιοθήκη. Τέλος, θα είναι υπεύθυνη για τον καθαρισμό της βιβλιοθήκης κατά την έξοδο του παιχνιδιού.

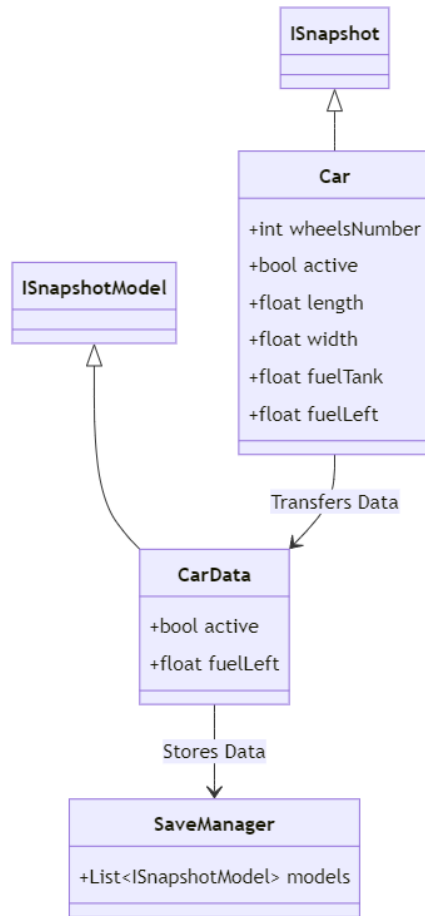
#### 5. ISnapshot interface

- a. Το ISnapshot interface θα είναι το πρώτο πιο σημαντικό component της αρχιτεκτονικής αφού θα είναι υπεύθυνο να ξεχωρίζει τις κλάσεις που προορίζονται για saving από τον Snapshot μηχανισμό. Θα περιέχει όλες τις απαραίτητες μεθόδους που θα πρέπει να υλοποιήσει μία κλάση ώστε να αποστείλει τα δεδομένα της στον κεντρικό Save Manager την στιγμή του saving αλλά και στην περίπτωση του loading να δεχθεί το αποθηκευμένο πακέτο δεδομένων ώστε να ξανά κάνει mutate τα serialized class fields του που είχαν αποθηκευτεί. Στη συνέχεια, θα περιέχει μεθόδους για την ανάκτηση των referenced SMRIs του ώστε μέσω του Save Manager να ανακτήσει τα references στα objects που είχε κατά τη διάρκεια του save. Τέλος, θα περιέχει και ένα field για το SMRI της ίδιας της κλάσης.

#### 6. ISnapshot Model interface

- a. Το ISnapshot Model interface θα είναι το δεύτερο πιο σημαντικό component της αρχιτεκτονικής αφού θα είναι υπεύθυνο να ξεχωρίζει τα λεγόμενα “μοντέλα των ISnapshot” που θα περιέχουν τα δεδομένα προς αποθήκευση στη βιβλιοθήκη. Θα περιέχει δύο fields, ένα για το SMRI της κλάσης που αντιπροσωπεύει και ένα δεύτερο για τα Referenced SMRIs που θα έχει η κλάση αυτή. Ουσιαστικά, θα γίνεται inherit από απλά data carriage structs.

Η σχέση των Save Manager, ISnapshot και ISnapshotModel μπορεί να διακριθεί στην Εικόνα 19 και στην Εικόνα 20.



Εικόνα 20: Αρχιτεκτονική σχέση μεταξύ Save Manager, ISnapshot και ISnapshotModel

### 3.10 Σύγχρονη ή Ασύγχρονη προσέγγιση;

Όλες οι μέθοδοι της βιβλιοθήκης θα λειτουργούν στο main thread της εφαρμογής, εκτός αν η host γλώσσα κάνει χρήση των μεθόδων της σε ένα καινούργιο thread, όπου σε εκείνη την περίπτωση οι λειτουργίες της βιβλιοθήκης θα λειτουργούν ασύγχρονα σε νέο thread αλλά θα τρέχουν σύγχρονα με βάση την σειρά που καλούνται στο εκάστοτε thread. Αυτή η απόφαση πάρθηκε έτσι ώστε να μην υπάρχουν περιορισμοί στην χρήση της βιβλιοθήκης από κάποια άλλη γλώσσα και η απόφαση αυτή θα βρίσκεται στην κρίση του εκάστοτε developer που κάνει χρήση της βιβλιοθήκης.

### 3.11 Βαθμός πραγματοποίησης στόχων διατριβής

@TODO: Μετά την εισαγωγή

### 3.12 Ανάδειξη σχεδιαστικών στοιχείων

Παρόλο που η βιβλιοθήκη και τα επί μέρους τμήματα της πτυχιακής αυτής έχουν σχεδιαστεί με βασικές πρακτικές του Object Oriented Design, το ζήτημα του serialization των object references

παρέμεινε. Έτσι, τα SMRIs και το containerization των δεδομένων εσωτερικά και εξωτερικά της βιβλιοθήκης σχεδιάστηκαν για να μπορέσουν τα δεδομένα προς αποθήκευση να παραμείνουν απaráλλακτα κατά τη μεταφορά τους. Επιπλέον, όπως έχει προαναφερθεί στα προηγούμενα κεφάλαια, τα SMRIs δρουν σαν απλοί αριθμοί που συσχετίζουν ένα πακέτο δεδομένων με την κλάση από την οποία ήρθαν και έτσι αποθηκεύοντας το SMRI μίας κλάσης αλλά μαζί και τα SMRI των referenced objects που έχει αυτή η κλάση, μπορεί μετά ένας μηχανισμός εύκολα και με τη χρήση μίας κεντρικής δομής να αναθέσει ξανά τα ίδια references στις κλάσεις που ήταν κατά τη διάρκεια της αποθήκευσης.

Στη συνέχεια, η συγκεκριμένη βιβλιοθήκη προορίζεται να δημοσιευθεί σαν open source library, οπότε ο component-driven σχεδιασμός της, θα επιτρέπει στον εκάστοτε προγραμματιστή να αλλάξει εύκολα μέσω του source κώδικα της διαφορετικά μέρη της βιβλιοθήκης όπως ο εσωτερικός serializer, πράγματα σχετικά με την ονοματοδοσία του τελικού αρχείου ή και ακόμα να την επεκτείνει εύκολα μέσω μίας Utilities κλάσης που θα περιέχει βοηθητικές μεθόδους για χρήση εσωτερικά της βιβλιοθήκης.

Τέλος, δύο επίσης σημαντικά χαρακτηριστικά, είναι πως οι μέθοδοι της θα δέχονται μόνο primitive type μεταβλητές και pointers κάτι που την καθιστά language agnostic και δεύτερων πως οι δομές που θα χρησιμοποιηθούν θα βασίζονται επάνω στα BigO metrics τους ώστε να μην υπάρχει μεγάλο performance overhead.

## Κεφάλαιο 4: Υλοποίηση

### 4.1 Εισαγωγή κεφαλαίου

@TODO

### 4.2 Υλοποίηση API και C++ DLL

Η βιβλιοθήκη δημιουργήθηκε και χωρίστηκε σε 6 συγκεκριμένα regions με αυτά να είναι τα:

1. Library Specific Structs
  - a. Τα structs Data και DataContainer χρησιμοποιούνται για το serialization των δεδομένων από το MsgPack και το containerization των δεδομένων αντίστοιχα.
2. GlobalVariables
  - a. Περιέχει σημαντικές μεταβλητές για την χρήση τους εσωτερικά της βιβλιοθήκης αλλά και την βασική δομή αποθήκευσης τύπου `std::map<unsigned int, DataContainer>` όπου είναι υπεύθυνη για το caching των πακέτων δεδομένων.
3. SavePath
  - a. Περιέχει τις mutator και accessor μεθόδους για τη σωστή διαχείριση του directory absolute save path.
4. SMRI Handling

- a. Περιέχει όλες τις μεθόδους για τη διαχείριση των SMRI εξωτερικά της βιβλιοθήκης αλλά και τη δυνατότητα διαγραφής ενός πακέτου δεδομένων με την εισαγωγή του SMRI του σαν παράμετρο.
- 5. DataCaching\_Packing
  - a. Περιέχει όλες τις μεθόδους για τη διαχείριση των αποθηκευμένων δεδομένων αλλά και την μέθοδο packData που ξεκινάει τη διαδικασία serialization των δεδομένων της βιβλιοθήκης.
- 6. LoadFromFile\_Unpacking
  - a. Περιέχει όλες τις μεθόδους για access και mutate του αρχείου που θα πρέπει να φορτώσει η βιβλιοθήκη ώστε να το κάνει deserialize και την βασική μέθοδο deserialization ονόματη unpackData.
- 7. DLL\_Cleanup
  - a. Περιέχει τις μεθόδους εκκαθάρισης της μνήμης του DLL.
- 8. LibraryUtils file
  - a. Περιέχει utility μεθόδους που χρησιμοποιούνται για string formatting και path validation εσωτερικά της βιβλιοθήκης.

#### **4.2.1 Library specific structs**

Το struct DataContainer έχει ως μοναδικό σκοπό το caching των δεδομένων του εισαγόμενου SMRI και των Referenced SMRI του. Περιέχει επίσης και τη template μέθοδο του MsgPack serializer ώστε να μπορεί το header file να κάνει το απαραίτητο serialization. Είναι άξιο να σημειωθεί, πως τα values των εισαγόμενων δεδομένων αποθηκεύονται σαν unsigned chars και όχι σαν unsigned char\* (pointer) και αυτό συμβαίνει διότι όπως είναι φυσικό δεν μπορεί μία θέση μνήμης να γίνει serialize αλλά πρέπει να γίνουν serialized τα values αυτών των θέσεων μνήμης.

```

9      /*
10     A struct for incoming data handling.
11     Used solely for data caching, packing and unpacking.
12     */
13     struct DataContainer
14     {
15         /*The saved data cache parent SMRI*/
16         unsigned int _Smri;
17         /*The cached data array size*/
18         int _DataSize;
19         /*The cached data converted in unsigned char for serialization*/
20         std::vector<unsigned char> _DataValues;
21         /*The referenced SMRI values this SMRI has.*/
22         std::vector<int> _RefSmris;
23
24         /*Packing msgpack::cppack method*/
25         template<class T>
26         void pack(T& pack) {
27             pack(_Smri, _DataSize, _DataValues, _RefSmris);
28         }
29     };

```

Εικόνα 21: Library source code, struct DataContainer

Το struct Data δημιουργήθηκε μόνο για τη διευκόλυνση της διαδικασίας του packing και του unpacking από το MsgPack.

```

31     /*
32     A struct used solely for packing and unpacking purposes.
33     Helps in making DataContainer serialization and deserialization easier.
34     */
35     struct Data
36     {
37         /*The map containing the smri-data pairs meant for serialization and deserialization.*/
38         std::map<unsigned int, DataContainer> _ModelsCache;
39
40         /*Packing msgpack::cppack method*/
41         template<class T>
42         void pack(T& pack) {
43             pack(_ModelsCache);
44         }
45     };

```

Εικόνα 22: Library source code, struct Data

## 4.2.2 Region Global Variables

Στο region αυτό διακρίνεται η βασική δομή αποθήκευσης τύπου `std::map<unsigned int, DataContainer> _ModelsCache` στο οποίο το key είναι το εισαγόμενο SMRI και το value είναι το δημιουργημένο instance τύπου `DataContainer` όπου περιέχει τα αντιγραφμένα δεδομένα προς αποθήκευση. Επιλέχθηκε η δομή `std::map` το οποίο στην πραγματικότητα είναι ένα Red-Black Tree και



στα Big O metrics του έχει average αλλά και worst case scenarios, time complexity  $\theta(\log(n))$  και  $O(\log(n))$  αντίστοιχα. Το space complexity του φυσικά παραμένει  $O(n)$ .

Τέλος, οι constant μεταβλητές SAVE\_FORMAT και SAVE\_EXTENSION μπορούν να αλλαχθούν χωρίς κάποια επίπτωση στις υπόλοιπες λειτουργίες της βιβλιοθήκης με τη χρήση της LibraryUtils βιβλιοθήκης.

```
47  ✓ #pragma region GlobalVariables
48      /*The save format of the saved files.*/
49      const std::string SAVE_FORMAT = "{date}_{cnt}";
50      /*The save file extension*/
51      const std::string SAVE_EXTENSION = ".sav";
52  ✓ /*
53      The global SMRI value used in data storage and reference preservation.
54      Default value is -1
55      */
56      int _GlobalSmriValue = -1;
57      /*A map storing SMRI-Data pairs. Used in serialization and deserialization.*/
58      std::map<unsigned int, DataContainer> _ModelsCache;
59      /*The externally-set absolute save path.*/
60      std::string _SavePath = "";
61      /*The externally-set file name to unpack from.*/
62      std::string _LoadFile = "";
63  #pragma endregion
```

Εικόνα 23: Library source code, GlobalVariables region

### 4.2.3 Region Save Path