1. Benchmark des sérialisations

Lors de la confection d'un binding capturant les données de capteurs pour bateaux connectées (SP70C radar, WIT GPS), nous avons détecté un soucis de performance. En effet, ce binding devait envoyer les données des capteurs au binding 'signal composer', ce dernier muni d'un plugin en adéquation. Le capteur SP70C donne des informations toutes les 20ms, nous espérions donc avoir des données en sortie du 'signal composer' à cette fréquence. En pratique ce n'était pas le cas, le débugger nous informait du warning « too many job ». En y faisant un profilage via l'outil open-source valgrind sur le lancement de l' afb-daemon, nous remarquons que plus de 85 % du processus était pris par la librairie JSON-C qui s'occupait de sérialiser et de parser les données intra-processus (entre binding).

De là est venue la question suivante, est-ce qu'il ne serait pas mieux de changer la techno utilisé pour le parssage des données ? (Cette question était également ressortie plus tôt mais n'a pas été relevé puisqu' aucune erreur n'était survenue).

1.1 Techno de sérialisation testé

Les techno testées dans cette étude sont les suivantes :

- JSON via la librairie JSON-C,

- JSON via la librairie FASTJSON (forqué de JSON-C et allégé),

- JSON via sa représentation en string,

- CBOR,

- PROTOBUF,

- XDR.

Pour la prochaine version des bindings (Version 4), le souhait de garder le JSON a été évoqué. Ce que l'on s'intéressera en particulier ici sera donc les différences entre les librairies utilisant le JSON.

A savoir que nous avons deux moyens pour transformer nos données en JSON. Soit nous utilisons des objets JSON classique, ainsi la représentation interne d'une structure de structure de données sera comme cette exemple :

{'structure 1' : {'data1' :'value1', 'data2' :'value2'}, 'structure2':{…},…}

Soit nous utilisons des tableaux de tableaux, comme ci-dessous :

[[value1, value2], […],…]

C'est pour cette raison que dans l'étude, pour chaque librairie de JSON, un test est effectué en MAPPANT (option MAP)et un autre est effectué en TABLANT (option ARRAY).

1.2 Présentation des résultats

Sans plus attendre, voici les résultats obtenues par le benchmark (clonable suivant le lien : http://git.ovh.iot/vlefebvre/benchmark-serialization).

1.2.1 Comparaison du temps

Ici, nous comparons le temps que prends une techno à sérialiser puis à parser 1 millions de structure en C (le type de structure est celle représenté sur le Readme.md du projet, c'est à dire la structure C utilisé dans le binding SP70C).

Pour chaque Techno une référence en C est simulée, il consiste à effectuer un premier memcpy dans un buffer en mémoire, puis d'en effectuer un second. Simulant ainsi la sérialisation puis la parsage des données. La dernière colonne du tableau ci-dessous représentant le facteur entre le temps pris par la référence et le temps pris pour la techno, de sorte à mieux se rendre compte des différences.

Techno	TIMER data: 1 000 000 struct C		
	for the techno	for the C reference	Without Ref:
	(ms)	(ms)	(ms)
JSON-C : MAP	6277,830196	1200,892394	5076,937802
JSON-C : ARRAY	2998,7211	1191,661614	1807,059486
FASTJSON : MAP	5352,582007	1139,514328	4213,067679
FASTJSON : ARRAY	3949,364817	1200,40885	2748,955967
STRINGJSON : MAP	9230,529687	1210,041258	8020,488429
STRINGJSON : ARRAY	4840,940771	1208,35273	3632,588041
CBOR : MAP	4157,415162	1148,408449	3009,006713
CBOR : ARRAY	2961,493036	1138,149888	1823,343148
PROTOBUF	2592,73813	1140,633496	1452,104634
XDR	1514,04666	1137,758169	376,288491

1.2.2 Comparaison du pourcentage de CPU pris par l'USER

Ici, nous comparons le pourcentage qu' utilise une techno à sérialiser puis à parser 100 000 structures C qui sont envoyé à une certaine fréquence. La fréquence testé ,qui est proche de la limite pour que 100 % des données soit transférées, est fixée à 50kHz (donnée toutes les 20µs).

Le programme va créer deux threads, un qui va prendre une structure C, la sérialiser puis la stocker dans un buffer mémoire. L'autre va récupérer dans le buffer la donnée sérialisée pour ensuite la stocker en structure C. Des sémaphores sont implémentés pour bien synchroniser les tâches et éviter des fuites de mémoires. Il a été également prévue que si le temps de sérialisation puis de parsage est plus long que la fréquence d'envois de données, alors la donnée est manqué et à la fin nous pouvons voir le pourcentage de données bien transférer (étude suivante).

Techno	CPU PERCENTAGE (USER) data: 100 000 / frequency : 20µs		
	for the techno	for the C reference	Without Ref:
	(%)	(%)	(%)
JSON-C : MAP	15,9721	8,54256	7,42954
JSON-C : ARRAY	13,6176	9,7356	3,882
FASTJSON : MAP	17,2202	7,7794	9,4408
FASTJSON : ARRAY	15,2671	8,2157	7,0514
STRINGJSON : MAP	18,1587	9,9473	8,2114
STRINGJSON : ARRAY	14,3041	7,8589	6,4452
CBOR : MAP	13,9631	9,8442	4,1189
CBOR : ARRAY	12,4146	9,7576	2,657
PROTOBUF	12,2676	9,1648	3,1028
XDR	9,8269	9,8547	-0,027799999999999

1.2.3 Comparaison des données manquées

La même étude est effectué mais cette fois-ci avec une fréquence plus soutenues de 10µs (100kHz). Et on s'interresse cette fois-ci aux donnée manquées :

Techno	CPU PERCENTAGE (USER) data: 100 000 / frequency : 10µs		
	for the techno	for the C reference	percentage of data lost
	(%)	(%)	(%)
JSON-C : MAP	16,2015	8,1891	10
JSON-C : ARRAY	13,037	11,3055	1
FASTJSON : MAP	18,2595	9,5965	8
FASTJSON : ARRAY	14,522	8,2352	2
STRINGJSON : MAP	17,8897	8,8097	15
STRINGJSON : ARRAY	15,5686	8,1932	3
CBOR : MAP	15,1908	9,7744	5
CBOR : ARRAY	11,5144	8,6803	0
PROTOBUF	11,4677	8,202	0
XDR	11,1945	8,4452	0

1.3 Conclusion / Voies d'amélioration du Benchmark

Au vue des résultats, nous pouvons remarquer que si nous gardons la sérialisation en JSON il n'y auras aucune différence marquante entre les différentes librairies sauf si nous choisissons de représenter les données en formats tableaux ! De cette façon, le choix de la librairie JSON-C n'est pas une mauvaise idée. Mais faisant ainsi nous serions obligés de connaître la structure des données lors de l'envoie mais aussi lors de la réception.

Voies d'amélioration du projet :

Ce projet mérite d'être encore à améliorer, notamment pour la sérialisation en JSON STRING, sous la forme de tableau,, car il a été fait à la main, et une optimisation est possible. Cela envisagerait néanmoins d'écrire une nouvelle API.

La librairie JSON-C++ de nlohmann est également implémentée, cependant ayant des résultats assez alarmant, je ne les ai volontairement pas ressorti. Des fuites de mémoires pour cette techno sont encore présentes et je continue à y travailler dessus.

Il faudrait également pouvoir différencier les résultats entre la sérialisation et le parsing, avoir le temps brut de la sérialisation puis le temps brut du parsing.

Une autre amélioration est envisagée : effectuer beaucoup plus de simulations pour pouvoir ressortir un tableau montrant l'évolution du pourcentage du CPU pris en fonction de la fréquence d'envois des données.

Et enfin, y ajouter une étude sur l'échange du buffer entre les 2 process, et ainsi se rapprocher du cas réel.

Practices makes progress, not perfection.