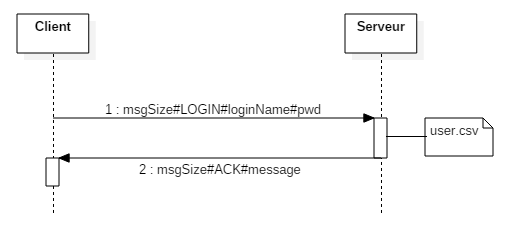
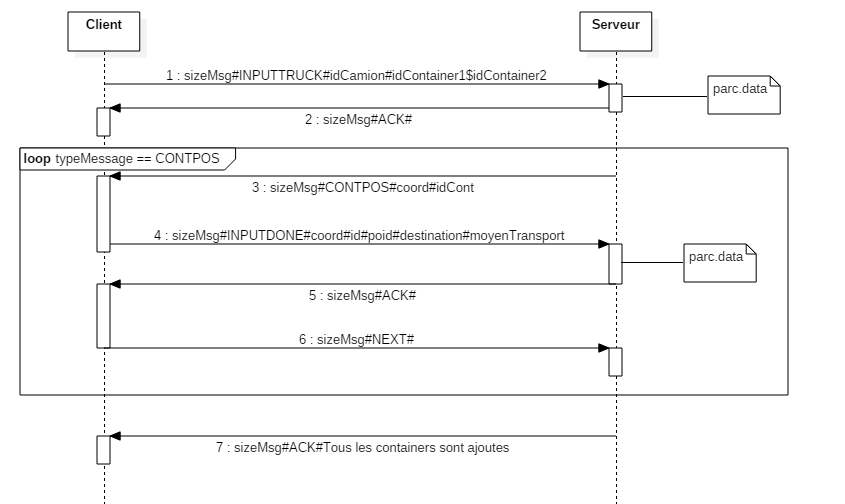
**Laboratoire Résaux et technologie de l’internet : Evaluation 1**

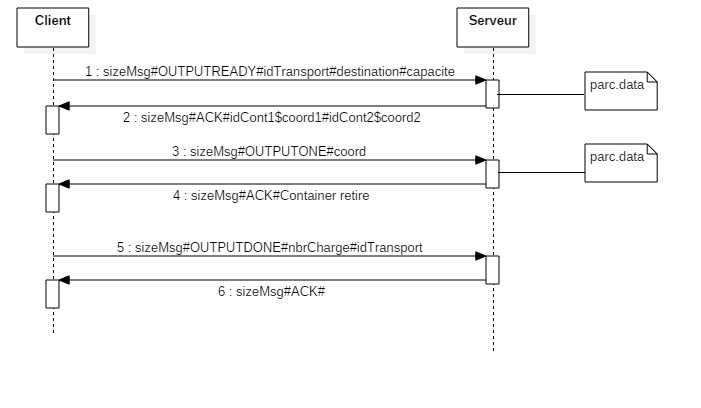
**Diagramme protocole CMMP :**

**Login client :**



**Input ready + input done**



**Output Ready + Output One + Output Done**

**Code serveur :**

1. #include <iostream>
2. #include <sys/socket.h>
3. #include <errno.h>
4. #include <netinet/in.h>
5. #include <netdb.h>
6. #include <cstdlib>
7. #include <cstring>
8. #include <pthread.h>
10. **using** **namespace** std;
12. #include "../Librairie/socket/socket.h"
13. #include "../Librairie/socket/socketServeur.h"
14. #include "../Librairie/fichierProp/fichierProp.h"
15. #include "../LibrairieConteneur/protocole.ini"
16. #include "../LibrairieConteneur/sendFunction.h"
17. #include "../Librairie/exceptions/errnoException.h"
18. #include "../Librairie/log/log.h"
19. #include "parc.h"
21. #define MAXCLIENT 2

24. pthread\_mutex\_t mutexIndiceCourant;
25. pthread\_cond\_t condIndiceCourant;
26. **int** indiceCourant = -1;
28. pthread\_mutex\_t mutexThreadsLibres;
29. pthread\_cond\_t condThreadsLibres;
30. **int** threadsLibres = MAXCLIENT;
32. pthread\_mutex\_t mutexLog;
33. Log logFile("logServeur.txt");
35. pthread\_mutex\_t mutexParc;
36. Parc parcFile("parc.dat");
38. pthread\_t threadsLances[MAXCLIENT];
39. Socket\* socketOuverte[MAXCLIENT];
41. **void**\* threadClient(**void**\* p);
42. **void** finConnexion(**int** cTraite, Socket\* s);
43. **int** login(Socket\* s, **int** clientTraite);
45. **void** inputTruck(Socket\*s, **int** clientTraite, string requete);
46. **void** inputDone(Socket\*s, **int** clientTraite, string listContainer, string listPosition);
48. **void** outputReady(Socket\* s, **int** clientTraite, string requete);
49. **void** outputOne(Socket\* s, **int** clientTraite);
51. **int** main()
52. {
53. //Lecture des informations dans le dossier properties
54. FichierProp fp = FichierProp("properties.txt");
56. string host = fp.getValue("HOST");
57. string port = fp.getValue("PORT");
58. string isip = fp.getValue("ISIP");
60. pthread\_cond\_init(&condIndiceCourant, NULL);
61. pthread\_mutex\_init(&mutexIndiceCourant, NULL);
62. pthread\_mutex\_init(&mutexLog, NULL);
63. pthread\_mutex\_init(&mutexParc, NULL);
65. SocketServeur\* sock = NULL; //On prÃ©pare une socket qui sera utilisÃ©e pour Ã©tablir la connection.
67. **try**
68. {
70. **if**(isip == "1")
71. sock = **new** SocketServeur(host , atoi(port.c\_str()), **true**); //Si dans le fichier prop on a une IP
72. **else**
73. sock = **new** SocketServeur(host , atoi(port.c\_str()), **false**); // Si dans le fichier prop on a un hostname
74. }
75. **catch**(ErrnoException er)
76. {
77. cout << er.getErrorCode() << "------" << er.getMessage() << endl;
78. exit(-1);
79. }
81. **for**(**int** i = 0; i < MAXCLIENT; i++)//On met la liste des sockets ouvertes Ã  NULL
82. socketOuverte[i] = NULL;
84. //LANCEMENT DES THREADS
85. **for**(**int** i = 0; i < MAXCLIENT; i++)
86. {
87. **int** ret = pthread\_create(&threadsLances[i], NULL, threadClient, (**void**\*) i);
89. pthread\_detach(threadsLances[i]);
90. }
92. //Choses sÃ©rieuses
93. **while**(1)
94. {
95. **try**
96. {
97. sock->ecouter(); //On se met Ã  l'Ã©coute d'une requÃªte cliente
98. }
99. **catch**(ErrnoException er)
100. {
101. cout << er.getErrorCode() << "------" << er.getMessage() << endl;
102. exit(-1);
103. }
105. pthread\_mutex\_lock(&mutexThreadsLibres);//On se met en attente sur une variable de condition : pas d'accept si pas de thread libre
106. **while**(threadsLibres == 0)
107. pthread\_cond\_wait(&condThreadsLibres, &mutexThreadsLibres);
109. **int** service = sock->accepter(); //On a un thread libre donc on peut accept
111. cout << "Nouveau client accepte " << endl;
113. **int** j;
115. **for**(j = 0; j < MAXCLIENT && socketOuverte[j] != NULL; j++); //On parcours nos thread pour trouver un libre
117. threadsLibres--;
118. pthread\_mutex\_unlock(&mutexThreadsLibres);
120. //section critique. On peut pas avoir deux threads qui lisent ces donnÃ©es en meme temps
121. pthread\_mutex\_lock(&mutexIndiceCourant);
122. socketOuverte[j] = **new** Socket(service); //creation d'une nouvelle socket service
123. indiceCourant=j; //on met la variable indice courant Ã  la position de la socket crÃ©Ã©e pour que le thread puisse savoir laquelle prendre
124. pthread\_mutex\_unlock(&mutexIndiceCourant);
125. pthread\_cond\_signal(&condIndiceCourant); //On rÃ©veil le thread au chÃ´mage
127. }
129. }
131. **void**\* threadClient(**void**\* p) //le thread lancÃ©
132. {
133. **int** requestType;
134. **while**(1)
135. {
136. pthread\_mutex\_lock(&mutexIndiceCourant);//On reste bloquÃ© ici tant qu'il n'y a pas de nouveau client (indice courant Ã  -1)
137. **while**(indiceCourant == -1)
138. pthread\_cond\_wait(&condIndiceCourant, &mutexIndiceCourant);
140. **int** clientTraite = indiceCourant; //on rÃ©cupÃ¨re l'indice de notre client dans le tableau de socket ouverte pour pas le perdre
141. indiceCourant = -1;//On remet Ã  -1 pour Ã©viter qu'un concurrent nous le pique.
143. Socket\* socketService = socketOuverte[clientTraite]; //recuperation de la socket du tableau
144. pthread\_mutex\_unlock(&mutexIndiceCourant);
146. **if**(!login(socketService, clientTraite))
147. **continue**;
149. **bool** cont = **true**;
151. **while**(cont) //boucle sur les demandes du client
152. {
154. string str = typeRequestParse(socketService->receiveChar(), &requestType);
156. **switch**(requestType)
157. {
158. **case** INPUT\_TRUCK:
159. inputTruck(socketService, clientTraite, str);
160. **break**;
161. **case** OUTPUT\_READY:
162. outputReady(socketService, clientTraite, str);
163. **break**;
164. **case** LOGOUT:
165. cont = **false**;
166. finConnexion(clientTraite, socketService);
167. **break**;
168. }
169. }
170. }
171. }
173. **void** finConnexion(**int** cTraite, Socket\* s) //On dÃ©connecte le client (on le fait pour LOGOUT ou en cas de problÃ¨me)
174. {
175. StructConnexion sc;
177. s->sendChar(composeConnexion(LOGOUT, sc));
178. s->finConnexion();
179. **delete** s;
181. pthread\_mutex\_lock(&mutexIndiceCourant);
182. socketOuverte[cTraite] = NULL;
183. pthread\_mutex\_unlock(&mutexIndiceCourant);
185. pthread\_mutex\_lock(&mutexThreadsLibres);
186. threadsLibres++;
187. pthread\_mutex\_unlock(&mutexThreadsLibres);
188. pthread\_cond\_signal(&condThreadsLibres);
189. }

192. **int** login(Socket\* s, **int** clientTraite)
193. {
194. string str;
195. **int** requestType;
196. **while**(1)
197. {
198. str = typeRequestParse(s->receiveChar(), &requestType);
200. **if**(requestType != LOGIN)
201. {
202. s->sendChar(composeAckErr(ERREUR, "INVALIDE"));
204. finConnexion(clientTraite, s);
205. **return** 0;
206. }
208. **if**(requestType == LOGIN)
209. {
210. StructConnexion sc;
211. FichierProp fp("login.csv", ';');
213. cout << "trame recue : " << str;
214. sc = parseConnexion(str);
216. string test = fp.getValue(sc.nom);
218. **if**(!test.compare("#"))
219. {
220. s->sendChar(composeAckErr(ERREUR, "LOGERR"));
221. }
222. **else**
223. {
224. cout << sc.motDePasse << endl;
225. **if**(!test.compare(sc.motDePasse))
226. {
227. s->sendChar(composeAckErr(ACK, "ALLRIGHT"));
228. **return** 1;
229. }
230. **else**
231. s->sendChar(composeAckErr(ERREUR, "LOGERR"));
233. }
234. }
235. **else** **if**(requestType == LOGOUT)
236. {
237. finConnexion(clientTraite, s);
238. **return** 0;
239. }
240. }
241. }

244. **void** inputTruck(Socket\*s, **int** clientTraite, string requete)
245. {
246. StructInputTruck sit = parseInputTruck(requete);
247. **char** \*lec, \*tok;
248. **char** sep = CONTAINER\_SEPARATION;
249. **char**\* saveptr;
250. string retPosition = "", ret = "";
251. **bool** erreur = **false**, cont = **true**;
253. lec =  **new** **char** [sit.idContainers.length()+1];
254. strcpy(lec, sit.idContainers.c\_str());
256. tok = strtok\_r(lec, &sep, &saveptr);
258. **while**(tok != NULL)
259. {
260. pthread\_mutex\_lock(&mutexParc);
261. ret = parcFile.getFirstFree(); //Renvois les coord du premier emplacement libre sous forme x;y
262. pthread\_mutex\_unlock(&mutexParc);
264. **if**(!ret.compare(""))
265. {
266. erreur = **true**;
267. **break**;
268. }
270. retPosition = retPosition + ret;
272. tok = strtok\_r(NULL, &sep, &saveptr);
274. **if**(tok != NULL)
275. retPosition = retPosition + CONTAINER\_SEPARATION;
277. }
279. **if**(erreur)
280. {
281. s->sendChar(composeAckErr(ERREUR, "Pas assez de place dans le parc"));
283. **if**(retPosition.compare(""))
284. {
285. pthread\_mutex\_lock(&mutexParc); //On libere les places qui Ã©taient rÃ©servÃ©es dans le fichier
286. parcFile.freeSpace(retPosition);
287. pthread\_mutex\_unlock(&mutexParc);
288. }
289. **return**;
290. }
291. **else**
292. {
293. s->sendChar(composeAckErr(ACK, ""));
294. inputDone(s, clientTraite, sit.idContainers, retPosition);
295. }
296. }

299. **void** inputDone(Socket\*s, **int** clientTraite, string listContainer, string listPosition)
300. {
301. **char** \*lecContainer, \*tokContainer, \*saveptrContainer, \*lecPosition, \*tokPosition, \*saveptrPosition;
302. **char** sep = CONTAINER\_SEPARATION;
304. lecContainer =  **new** **char** [listContainer.length()+1];
305. strcpy(lecContainer, listContainer.c\_str());
307. lecPosition =  **new** **char** [listPosition.length()+1];
308. strcpy(lecPosition, listPosition.c\_str());

311. tokContainer = strtok\_r(lecContainer, &sep, &saveptrContainer);
312. tokPosition = strtok\_r(lecPosition, &sep, &saveptrPosition);
314. **while**(tokContainer != NULL)
315. {
316. string str;
317. **int** requestType;
318. StructInputDone sid;
319. StructContainerPosition scp;
321. scp.coord = tokPosition;
322. scp.id = tokContainer;
324. s->sendChar(composeContPos(CONT\_POS, scp));
326. str = typeRequestParse(s->receiveChar(), &requestType);
328. **if**(requestType == INPUT\_DONE)
329. {
331. sid = parseInputDone(str);
333. **if**(sid.poids > 100)
334. {
335. pthread\_mutex\_lock(&mutexParc);
336. parcFile.freeSpace(sid.coord);
337. pthread\_mutex\_unlock(&mutexParc);
338. s->sendChar(composeAckErr(ERREUR, "Le container est trop lourd : enregistrement annule"));
339. }
340. **else**
341. {
342. pthread\_mutex\_lock(&mutexParc);
343. parcFile.placeContainer(sid);
344. pthread\_mutex\_unlock(&mutexParc);
345. s->sendChar(composeAckErr(ACK, "Le container a ete enregistre"));
346. }
348. }
349. **else**
350. {
351. pthread\_mutex\_lock(&mutexParc); //On libere les places qui Ã©taient rÃ©servÃ©es dans le fichier
352. parcFile.freeSpace(listPosition);
353. pthread\_mutex\_unlock(&mutexParc);
354. finConnexion(clientTraite, s);
355. **return**;
356. }
358. str = typeRequestParse(s->receiveChar(), &requestType);
360. **if**(requestType != NEXT)
361. {
362. finConnexion(clientTraite, s);
363. **return**;
364. }
366. tokContainer = strtok\_r(NULL, &sep, &saveptrContainer);
367. tokPosition = strtok\_r(NULL, &sep, &saveptrPosition);
369. }
371. s->sendChar(composeAckErr(ACK, "Tous les containers ont ete traites"));
373. }

376. **void** outputReady(Socket\* s, **int** clientTraite, string requete)
377. {
378. StructOuputReady sor  = parseOutputReady(requete);
380. pthread\_mutex\_lock(&mutexParc);
381. string listeContainer = parcFile.outputList(sor);
382. pthread\_mutex\_unlock(&mutexParc);
384. **if**(listeContainer.compare(""))
385. {
386. s->sendChar(composeAckErr(ACK, listeContainer));
387. outputOne(s, clientTraite);
388. }
389. **else**
390. s->sendChar(composeAckErr(ERREUR, "Aucun containers pour cette destination"));


394. }
396. **void** outputOne(Socket\* s, **int** clientTraite)
397. {
398. **int** requestType;
399. string reponse = typeRequestParse(s->receiveChar(), &requestType);
401. **while**(requestType == OUTPUT\_ONE)
402. {
403. StructOutputOne soo = parseOutputOne(reponse);

406. pthread\_mutex\_lock(&mutexParc);
407. parcFile.freeSpace(soo.emplacement);
408. pthread\_mutex\_unlock(&mutexParc);
410. cout << "retire du fichier" << endl;
412. s->sendChar(composeAckErr(ACK, "container retire"));
414. reponse = typeRequestParse(s->receiveChar(), &requestType);
415. }
417. **if**(requestType != OUTPUT\_DONE)
418. {
419. finConnexion(clientTraite, s);
420. }
422. s->sendChar(composeAckErr(ACK, "les containers ont bien ete retires"));
423. }