Analyzing the Request Log file

By Jean-Yves Fock-Hoon

Technical Note 06-04

Overview

4D Server has the ability to generate a log of requests made to the server (called the "request log file"). The purpose of this Technical Note is to demonstrate how the analysis of this file can be used to improve the performance of your database.

Introduction

As noted in Technical Note 06-03, "Recording Information sent Between 4D Client and 4D Server", the network is often the limiting factor of performance in a Client/Server environment. If the latency is high, client performance will be slower and can also be subject to more timeouts. If the latency is very low, such as on a LAN, the client performance can also be affected if too many of them are communicating with the server at the same time.

Obviously, the more requests the client makes, the more potential the server has to slowdown since it will need to answer to all of them. Additionally, if too many requests are sent and the latency is very bad, client performance may also be considerably decreased.

Note: Technical Note 06-03 showed how to open and display the request log file in a more understandable language. This Technical Note uses the same technique to view the request log file so it may be useful to review the previous Technical Note. This Technical Note uses a modified copy of the example database that was included in Technical Note 06-03.

Example 1: Browsing records

There are several techniques often used to browse the current selection in 4D. People from the "old school" might use the End Of File (EOF) technique, e.g.:

```
While (Not(End selection([Data])))
   NEXT RECORD([Data])
End while
```

Here a While loop us used with a call NEXT RECORD nested in the loop to traverse the selection.

Others prefer to use a For loop with a NEXT RECORD, e.g.:

```
$nbRecs:=Records in selection([Data])
For ($i;1;$NbRecs)
        NEXT RECORD([Data])
End for
```

Still others prefer a For loop with a GOTO RECORD or GOTO SELECTED RECORD instead of NEXT RECORD:

```
$NbRec:=Records in selection([Data])
For ($i;1;$NbRec)
        GOTO SELECTED RECORD([Data];$i)
End for
```

The request log file can be used to compare these three techniques.

Take a look at the first example, the EOF technique.

```
ALL RECORDS([Data])
While (Not(End selection([Data])))
   NEXT RECORD([Data])
End while
```

The log file data below is based on a selection of 3 records.

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	18	0
4	Rec_load	6	96076	0
5	Rec_Unload	6	6	0
6	Rec_load	6	96074	0
7	Rec_Unload	6	6	0
8	Rec_load	6	96076	0
9	Rec_Unload	6	6	0

This code generates 9 requests. The first 3 requests are generated by the call to ALL RECORDS. Each call to NEXT RECORD performs a Load and an Unload.

The next example demonstrates the use of NEXT RECORD inside a For loop.

```
ALL RECORDS([Data])
$nbRecs:=Records in selection([Data])
For ($i;1;$NbRecs)
        NEXT RECORD([Data])
End for
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	18	0
4	Rec_load	6	96076	0
5	Rec_Unload	6	6	0
6	Rec_load	6	96074	0
7	Rec_Unload	6	6	0
8	Rec_load	6	96076	0
9	Rec_Unload	6	6	0

Notice there is no difference when compared to the EOF technique. This code generates 9 requests. The first 3 requests are generated by the call to ALL RECORDS. Each call to NEXT RECORD performs a Load and an Unload.

The third example uses the GOTO SELECTED RECORD command within a For loop.

```
ALL RECORDS([Data])
$NbRec:=Records in selection([Data])
For ($i;1;$NbRec)
  GOTO SELECTED RECORD([Data];$i)
End for
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	18	0
4	Rec_Unload	6	6	0
5	Rec_load	6	96076	3
6	Rec_Unload	6	6	0
7	Rec_load	6	96076	3
8	Rec_Unload	6	6	0
9	Rec_load	6	96074	1
10	Rec_Unload	6	6	0
11	Rec_load	6	96076	3

Notice there are 2 more requests. This is because the call to ALL RECORDS will always load the first record. Then, in the loop, GOTO SELECTED RECORD is called on record 1, so the first record is reloaded.

The For loop is generally the most efficient loop since a selection can be empty or can contain only one record. A While or Repeat loop could be used but that would require more tests.

The last example uses a For loop with GOTO RECORD. When using GOTO RECORD a record number is needed. The record numbers of the selection are retrieved from a call to the SELECTION TO ARRAY command.

```
ALL RECORDS([Data])
SELECTION TO ARRAY([Data];$alrecnum)
For ($i;1;Size of array($alrecnum))
        GOTO RECORD([Data];$alrecnum{$i})
End for
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	18	0
4	Rec_load	6	96076	3
5	Sel_SelectionToArray	20	31	0
6	Struct_GetNbTablesAndFields	2	54	0
7	Rec_Unload	6	6	0
8	Struct_GetNbTablesAndFields	2	26	0
9	Rec_load	6	96076	3
10	Sel_ReduceToCurrentRec	10	6	0
11	Rec_Unload	6	6	0
12	Struct_GetNbTablesAndFields	2	26	0
13	Rec_load	6	96074	3
14	Sel_ReduceToCurrentRec	10	6	0
15	Rec_Unload	6	6	0
16	Struct_GetNbTablesAndFields	2	26	0
17	Rec_load	6	96076	0
18	Sel_ReduceToCurrentRec	10	6	0

In this case the number of requests has doubled. The first 3 requests are generated by the call to ALL RECORDS. The next 3 requests are generated by the call to SELECTION TO ARRAY. Each call to GOTO RECORD generates 4 requests: unload the current record; retrieve the structure; load the record; and define the current record.

Note that the GOTO RECORD command is a very fast way to access a record, faster than searching for the record, since the address is known. However in a Client/Server environment, it can become disastrous as it requires more requests, which could cause the whole system to slowdown, especially if the network is very active.

There is no real difference between a While and a For loop when using NEXT RECORD. The use of GOTO SELECTED RECORD can be considered acceptable. However the use of GOTO RECORD presents some serious performance risks since more requests will be generated.

Example 2: Creating a selection of records

In this example three records will be created.

The first technique simply performs a loop and uses the CREATE RECORD and SAVE RECORD commands.

```
C_TEXT($mb)
$mb:="A"*32000
For ($i;1;3)
   CREATE RECORD([Data])
   [Data]StringA:=String(Random)+"-"+String(Random)
   [Data]LongIntA:=Random
   [Data]RealA:=Random/10
   [Data]fText1:=$mb
   [Data]fText2:=$mb
   [Data]fText3:=$mb
   SAVE RECORD([Data])
End for
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Rec_Save	96074	10	0
2	Sel_ReduceToCurrentRec	10	6	0
3	Rec_Unload	6	6	0
4	Rec_Save	96074	10	0
5	Sel_ReduceToCurrentRec	10	6	0
6	Rec_Unload	6	6	0
7	Rec_Save	96074	10	0
8	Sel_ReduceToCurrentRec	10	6	0

Notice SAVE RECORD will reduce the selection to the current record. Therefore, the current record will be unloaded in order to set the new created record as current record. The Client has sent a total of 288,264 bytes to the Server.

The next example creates the records using ARRAY TO SELECTION.

```
C_TEXT($mb)
$mb:="A"*32000
ARRAY TEXT($as;3)
ARRAY TEXT($at1;3)
ARRAY TEXT($at2;3)
ARRAY TEXT($at3;3)
ARRAY LONGINT($al;3)
ARRAY REAL($ar;3)

For ($i;1;3)
   $as{$i}:=String(Random)+"-"+String(Random)
   $al{$i}:=Random
```

```
   $ar{$i}:=Random/10
   $at1{$i}:=$mb
   $at2{$i}:=$mb
   $at3{$i}:=$mb
End for

ARRAY TO
SELECTION($as;[Data]StringA;$al;[Data]LongIntA;$ar;[Data]RealA;$at1;[Data]fText1;$at
2;[Data]fText2;$at3;[Data]fText3)
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_ArrayToSelection	288201	6	2
2	Struct_GetNbTablesAndFields	2	26	0

Notice there are only two requests and the Client sent only 288,203 bytes. If multiple records need to be created at the same time it is best to ARRAY TO SELECTION. Of course, the use of arrays will require that both Client and Server have enough memory for the storage.

Example 3: Modifying a selection records

When it comes to modifying records in a batch process, a few methods can be used.

The first one is a simple loop on the selection with the use of SAVE RECORD and NEXT RECORD. In this example 8 records will be modified:

```
ALL RECORDS([Data])
While (Not(End selection([Data])))
   Data]LongIntA:=[Data]LongIntA+100-50-50
   SAVE RECORD([Data])
   NEXT RECORD([Data])
End while
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	38	0
4	Rec_load	6	96076	2
5	Rec_Save	96074	10	0
6	Rec_Unload	6	6	0
7	Rec_load	6	96074	3
8	Rec_Save	96072	10	1
9	Rec_Unload	6	6	0
10	Rec_load	6	96076	3

11	Rec_Save	96074	10	0
12	Rec_Unload	6	6	0
13	Rec_load	6	96076	3
14	Rec_Save	96074	10	0
15	Rec_Unload	6	6	0
16	Rec_load	6	96076	3
17	Rec_Save	96074	10	0
18	Rec_Unload	6	6	0
19	Rec_load	6	96076	3
20	Rec_Save	96074	10	0
21	Rec_Unload	6	6	0
22	Rec_load	6	96076	3
23	Rec_Save	96074	10	0
24	Rec_Unload	6	6	0
25	Rec_load	6	96076	3
26	Rec_Save	96074	10	0
27	Rec_Unload	6	6	0

Again, the 3 first requests are generated by ALL RECORDS. Then, for each record, there is a load, save and unload. Also, the record is transferred from the Server to the Client, the Client modifies the record and then sends it back to the Server. This generates 27 requests.

The next example uses APPLY TO SELECTION to modify the records:

```
ALL RECORDS([Data])
APPLY TO SELECTION([Data];[Data]LongIntA:=[Data]LongIntA+100-50-50)
```

Here is the request log data:

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	38	0
4	Rec_load	6	96076	1
5	Struct_SendAvailableAutoLink2S	32	6	0
6	Rec_Unload	6	6	0
7	Rec_LoadAndSendData	40	96080	3
8	Rec_Save	96074	10	0
9	Rec_Unload	6	6	0
10	Rec_LoadAndSendData	40	96080	3
11	Rec_Save	96074	10	0
12	Rec_Unload	6	6	0
13	Rec_LoadAndSendData	40	96080	3
14	Rec_Save	96074	10	0
15	Rec_Unload	6	6	0
16	Rec_LoadAndSendData	40	96080	3
17	Rec_Save	96074	10	0
18	Rec_Unload	6	6	0
19	Rec_LoadAndSendData	40	96080	3
20	Rec_Save	96074	10	0
21	Rec_Unload	6	6	0

22	Rec_LoadAndSendData	40	96080	3
23	Rec_Save	96074	10	1
24	Rec_Unload	6	6	0
25	Rec_LoadAndSendData	40	96080	0
26	Rec_Save	96074	10	0
27	Rec_Unload	6	6	0
28	Rec_LoadAndSendData	40	96080	1
29	Rec_Save	96074	10	0
30	Set_Delete	81	6	0
31	Set_Send	98	6	0
32	Rec_Unload	6	6	0

With 8 records, we can generate 32 records. We can still see that the whole record is sent from the Server to the Client. The number of requests is almost the same. The difference is these 4 requests: a check on relations, the LockedSet set that APPLY TO SELECTION always creates for locked records and the unloading of the last record.

If you really like to use APPLY TO SELECTION, another way would be to create a stored procedure that will perform the APPLY TO SELECTION.

```
$a:=Execute on server("M_ApplyFormula";1024*1024;"ApplyFormula")
DELAY PROCESS(Current process;0)

Repeat
   IDLE
Until (Not(Test semaphore("Apply")))
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Proc_ExecuteOnServer	76	10	130
2	Sem_Set	38	8	0

As we can see, only 2 calls will be performed. However, you also have to design a way to retrieve the Lockset. A stored procedure may also use a lot of CPU time. If the execution is faster on the server, other clients may be slowed down because the increased activity of the server.

Our last method would be to use arrays. We saw that ARRAY TO SELECTION will be faster than SAVE RECORD.
At some point, the data needs to be transferred to the Client. The advantage of SELECTION TO ARRAY is that we are not obliged to send all fields. ARRAY TO SELECTION will modify only the fields defined as parameter for an existing record. If the record does not exist, a new record will be created.

```
ARRAY LONGINT($al;0)
ALL RECORDS([Data])
SELECTION TO ARRAY([Data]LongIntA;$al)
For ($i;1;Size of array($al))
   $al{$i}:=$al{$i}+100-50-50
```

```
End for
ARRAY TO SELECTION($al;[Data]LongIntA)
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	38	0
4	Rec_load	6	96076	3
5	Sel_SelectionToArray	20	33	0
6	Struct_GetNbTablesAndFields	2	54	0
7	Rec_Unload	6	6	0
8	Sel_ArrayToSelection	37	6	4
9	Struct_GetNbTablesAndFields	2	26	0

As we can see, ALL RECORDS will generate our selection and load the first record. The SELECTION TO ARRAY will generate the only array that we need. 4D Client will modify that array and will send it back to the Server. We can see that SELECTION TO ARRAY unloads the current record. We perform a total of 9 calls and less data has been transferred. You will want to use this technique in your code if it is compatible with the design of your database.

Example 4: Transactions

In our example, we are going to create some records using SELECTION TO ARRAY inside a transaction. Our code would look as follows:

```
START TRANSACTION
C_TEXT($mb)
$mb:="A"*32000
ARRAY TEXT($as;3)
ARRAY TEXT($at1;3)
ARRAY TEXT($at2;3)
ARRAY TEXT($at3;3)
ARRAY LONGINT($al;3)
ARRAY REAL($ar;3)

For ($i;1;3)
   $as{$i}:=String(Random)+"-"+String(Random)
   $al{$i}:=Random
   $ar{$i}:=Random/10
   $at1{$i}:=$mb
   $at2{$i}:=$mb
   $at3{$i}:=$mb
End for

ARRAY TO
SELECTION($as;[Data]StringA;$al;[Data]LongIntA;$ar;[Data]RealA;$at1;[Data]fText1;$at2;[Data]
fText2;$at3;[Data]fText3)
```

VALIDATE TRANSACTION

Order	Request Name	Bytes In	Bytes Out	Duration
1	Trans_Start	2	6	0
2	Sel_ArrayToSelection	288203	6	3
3	Trans_GetNbNewRecInside	2	10	0
4	Struct_GetNbTablesAndFields	2	26	0
5	Trans_Validate	20	26	1

As you can see, transactions will generate only 4 small requests. The use of transactions should not penalize your connections, unless your code generates one million of transactions per second…

Example 5: Sorting a selection

Our next example sort a selection of records. The number of records to be sorted does not matter since we're interested in requests that could be sent during the execution of the code.

ALL RECORDS([Data])
ORDER BY([Data];[Data]LongIntA;[Data]RealA;[Data]StringA)

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sel_AllRecords	2	10	0
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	262	0
4	Rec_load	6	50	0
5	Sort	292	6	12
6	Struct_GetNbTablesAndFields	2	26	0
7	Sel_CacheSelection	10	262	0
8	Rec_Unload	6	6	0
9	Rec_load	6	52	0

Our same 4 requests are generated for the ALL RECORDS while the ORDER BY will generate 5 additional requests. The first request will be the sort. The next 2 requests retrieve the new selection. The former record will be unloaded while the new first record is loaded.

Example 6: Searching a selection

In this example, let's perform a simple query as described below:

QUERY([Data];[Data]StringA="1@";*)
QUERY([Data]; | [Data]StringA="2@";*)
QUERY([Data]; | [Data]StringA="3@";*)
QUERY([Data]; | [Data]StringA="4@";*)
QUERY([Data]; | [Data]StringA="5@";*)
QUERY([Data]; | [Data]StringA="6@";*)

```
QUERY([Data]; | [Data]StringA="7@";*)
QUERY([Data]; | [Data]StringA="8@";*)
QUERY([Data]; | [Data]StringA="9@")
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Sem_Clear	38	6	0
2	Search	868	22	0
3	PRes_Write	24	6	0
4	Sel_CacheSelection	10	262	0
5	PRes_Write	24	6	0
6	Rec_Unload	6	6	0
7	PRes_Write	24	6	0
8	Rec_load	6	50	0
9	PRes_Write	24	6	0

As we can see, 9 requests are generated with a total of 994 bytes in and 314 bytes out. Since version 6.5, there is a command that can be used to match our example: QUERY WITH ARRAY. Let's see what would be the results.

```
ARRAY TEXT($at;10)
$at{1}:="1@"
$at{2}:="2@"
$at{3}:="3@"
$at{4}:="4@"
$at{5}:="5@"
$at{6}:="6@"
$at{7}:="7@"
$at{8}:="8@"
$at{9}:="9@"
QUERY WITH ARRAY([Data]StringA;$at)
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Search_QueryWithArray	65	10	1
2	Struct_GetNbTablesAndFields	2	26	0
3	Sel_CacheSelection	10	262	0
4	Rec_load	6	50	0

This command will generate 4 requests only, with only 83 bytes in and 348 bytes out. If you need to perform such queries, QUERY WITH ARRAY definitively offers a good alternative.

Example 7: Transferring variables in C/S mode

In this example, we are just going to retrieve some variables from 4D Server by using GET PROCESS VARIABLE command. The 2 variables would be a string array and a text array. Both inter-process arrays have been defined on

the Server and contain the same contains, i.e. the same string of 20 characters and have the same number of items.

```
ARRAY STRING(80;as;20)
GET PROCESS VARIABLE(-1;<>as;as)
ARRAY TEXT(at;0)
GET PROCESS VARIABLE(-1;<>at;at)
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Proc_GetProcessVar	50	8347	0
2	Proc_GetProcessVar	50	2265	0

As we can see, one request has been generated for each GET PROCESS VARIABLE. However, the size of bytes received is not the same. This is just a reminder that string arrays require more space than text arrays. This has also an impact when they need to be transferred between 4D Client and 4D Server. As you may know, you cannot pass arrays as parameters when creating stored procedure for example. Your workaround would be to drop the array into a BLOB and send the blob as parameter. In that case, the BLOB that contains the string array is going to be bigger than the other one; the transfer over the network will therefore be longer.

Example 8: Semaphores

This last example will look at one of the most common piece of code that everybody is using when dealing with semaphores.

```
$a:=Execute on server("P_Idle";1024*1024;"Idle")
DELAY PROCESS(Current process;60)
While (Test semaphore("Wait "))
   IDLE
End while
```

In our method, we perform a DELAY PROCESS in order to give time to 4D Server to create the stored procedure and define the semaphore. This is acceptable since the stored procedure will take more than 1 second. The P_Idle method looks as follows:

```
$a:=Semaphore("Wait ")
For ($i;1;50000)
   $ab:="a"*2000
   $ab:="a"*2000
   $ab:="a"*2000
   $ab:="a"*2000
   $ab:="a"*2000
End for
CLEAR SEMAPHORE("Wait ")
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Rec_LoadForModifyDisplaySelect	60	838	0
2	Rec_RecInTable	2	10	0
3	Proc_ExecuteOnServer	76	10	48
4	Rec_LoadForModifyDisplaySelect	60	838	0
5	Rec_RecInTable	2	10	0
6	Rec_RecInTable	2	10	0
7	Rec_RecInTable	2	10	0
8	Rec_RecInTable	2	10	0
9	Rec_RecInTable	2	10	0
10	Rec_RecInTable	2	10	0
11	Rec_RecInTable	2	10	0
12	Rec_RecInTable	2	10	0
13	Sem_Set	38	8	0
14	Sem_Set	38	8	0
15	Sem_Set	38	8	0
16	Sem_Set	38	8	0
17	Sem_Set	38	8	0
18	Sem_Set	38	8	0
…				
19097	Sem_Set	38	8	0
19098	Sem_Set	38	8	0
19099	Sem_Set	38	8	0
19100	Sem_Set	38	8	0
19101	Sem_Set	38	8	0
19102	Sem_Set	38	38	0

As we can see, more than 19000 Semaphore requests are sent to the Server. The server has to respond to each of them. This may decrease your performance CPU-wise or network-wise.

Let’s now have a look to another alternative. The code is the same except that IDLE is replaced by DELAY PROCESS (0) .

```
$a:=Execute on server("P_Idle";1024*1024;"Idle")
DELAY PROCESS(Current process;60)
While (Test semaphore("wait"))
   DELAY PROCESS(Current process;0)
End while
```

Order	Request Name	Bytes In	Bytes Out	Duration
1	Proc_ExecuteOnServer	76	10	162
2	Sem_Set	38	8	0
3	Sem_Set	38	8	0
4	Sem_Set	38	8	0
5	Sem_Set	38	8	0
6	Sem_Set	38	8	0
7	Sem_Set	38	8	0
8	Sem_Set	38	8	0
9	Sem_Set	38	8	0
10	Sem_Set	38	8	0

...				
12704	Sem_Set	38	8	0
12705	Sem_Set	38	8	0
12706	Sem_Set	38	8	0
12707	Sem_Set	38	8	0
12708	Sem_Set	38	8	0
12709	Sem_Set	38	38	0

As we can see, the number of request has dropped considerably. We went from almost 19102 to 12709.

Here is the number of requests sent by 4D Client to 4D Server per second:

IDLE		DELAY PROCESS		
Time	Request/second		Time	Request/second
11:25:32	784	11:27:43	2	
11:25:33	2664	11:27:44	2041	
11:25:34	2052	11:27:45	972	
11:25:35	831	11:27:46	1902	
11:25:36	866	11:27:47	998	
11:25:37	2512	11:27:48	1360	
11:25:38	1574	11:27:49	1164	
11:25:39	2518	11:27:50	1164	
11:25:40	1472	11:27:51	1320	
11:25:41	2092	11:27:52	1131	
11:25:42	1737	11:27:53	655	

As we can see, the average of number of requests is higher when using IDLE and the time taken to execute that stored procedure is the same, i.e. 10 seconds.

The IDLE command does not force an idle. It checks if an idle can be performed. An idle is performed if the process used at least one tick. In our first example, a check on the semaphore is performed, then, an idle request is performed. Since our process did not use one tick, the process will keep the hand and check the semaphore again, back and forth, until the process uses one tick. With DELAY PROCESS, an idle is forced and the hand is given to the next process on our 4D Client, then, back to our current process. DELAY PROCESS (0) does not delay the process but just forces an idle, while DELAY PROCESS (1) delays the process for 1 tick and automatically performs an idle.

Summary

By executing a few commands, we know now what type of request has been sent, how many and - sometimes - when 4D reloads or unloads the current record. With this information, we can now have an idea on how to optimize

the communication between Client and Server. In other words, reducing the communication between 4D Server and all 4D Clients may help 4D Server save some CPU time. 4D Server can then use that extra CPU time to handle more connections or simply doing its job faster. Of course, with less traffic, it can also help alleviate the load on the network.

However, reducing the number of requests is not necessarily a guarantee of speed. If, in all cases we saw, the speed gains are significant, keep in mind that a client server system is complex and may not yield such results every time.