Analyzing the Request Log file

By Jean-Yves Fock-Hoon

TN 06-04

Overview

4D Server には、クライアントから送信されたリクエストのログファイルを作成するという機能がついています。このテクニカルノートでは、4D Server のリクエストログファイルを解析し、データベースのパフォーマンスを改善するために活用する方法が説明されています。

Introduction

テクニカルノート 06-03 Recording Information sent Between 4D Client and 4D Server でも触れたように、ネットワークはクライアント-サーバ環境でしばしばパフォーマンスを左右する要因になります。ネットワークが混雑していれば、クライアントのパフォーマンスが低下し、タイムアウトも頻繁に起こるようになります。LAN などのように、ネットーワークが混雑していない場合、サーバに多数のクライアントがアクセスする結果として、やはりパフォーマンスが低下するかもしれません。

クライアントから多くのリクエストが送られるのであれば、当然のこととして、サーバはそれに対応できるだけの時間的余裕を必要とします。ネットワークが混雑している場合、クライアントのパフォーマンスはさらに低下することになります。

Note:テクニカルノート 06-03 では、リクエストログファイルを開いて分かりやすい形式で表示する方法を紹介しました。このテクニカルノートでも同じ手法を用いているので、以前のテクニカルノートを参照すると良いかもしれません。本テクニカルノートでは、06-03 で使用されたものを改造したサンプルデータベースを使用しています。

Example 1: Browsing records

4D でカレントセレクションを解析する方法はいろいろありますが、古いデベロッパであれば、いわゆる EOF(End Of File)技法を使用するかもしれません。

例：

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

上記の例では、While ループで NEXT RECORD コマンドを囲むことによって、セレクションの中を巡るようにしています。

代わりに For ループを使用することもできます：

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

ループの方法は同じでも、NEXT RECORD の代わりに GOTO RECORD や GOTO SELECTED RECORD コマンドを使用するパターンもあります：

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

3 種類の方法を比較するために、リクエストログファイルを解析してみましょう。

まずは最初の方法です。

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

下記は 3 レコードのセレクションで実行したリクエストログです：

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

このコードでは 9 件のリクエストが送信されました。最初の 3 件は ALL RECORDS のコールです。その後は NEXT RECORD がコールされるたびにロードとアンロードが繰り返されています。

次に For ループを使用する例について調べてみましょう。

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

リクエストログは次のようなものでした：

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

最初の例と同じように 9 件のリクエストが発生しました。最初の 3 件は ALL RECORDS のコールです。その後は NEXT RECORD がコールされるたびにレコードのロードとアンロードが繰り返されています。

次に GOTO SELECTED RECORD コマンドを使用する例について調べてみましょう。

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

リクエストログは次のようになりました：

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
1 0	Rec_Unload	6	6	0
1 1	Rec_load	6	96076	3

リクエストが 2 件増えている点に注目することができます。これは ALL RECORDS で最初のレコードがロードされているにも関わらず GOTO SELECTED RECORD で再びそのレコードがロードされているためです。

セレクションが 0 件あるいは 1 件のレコードである場合のことを考えれば、ここでは For ループがもっとも効率的な方法であると思われます。While ループ、Repeat ループとの差について検証するためには、さらにテストが必要です。

最後に GOTO RECORD コマンドについて調べてみましょう。GOTO RECORD コマンドではレコード番号を使用します。セレクションのレコード番号は SELECTION TO ARRAY コマンドで取得することができます。

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

リクエストログは次のようなものでした：

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
1 0	Sel_ReduceToCurrentRec	10	6	0
1 1	Rec_Unload	6	6	0
1 2	Struct_GetNbTablesAndFields	2	26	0
1 3	Rec_load	6	96074	3
1 4	Sel_ReduceToCurrentRec	10	6	0
1 5	Rec_Unload	6	6	0
1 6	Struct_GetNbTablesAndFields	2	26	0
1 7	Rec_load	6	96076	0
1 8	Sel_ReduceToCurrentRec	1 0	6	0

リクエストの数がなんと倍増してしまいました。最初の 3 件は ALL RECORDS のコールです。続く 3 件は SELECTION TO ARRAY のコールです。GOTO RECORD がコールされるたびに 4 件のリクエストが発生し、カレントレコードのアンロード、ストラクチャ情報の受信、レコードのロード、カレントレコードの定義が繰り返されています。

GOTO RECORD は、本来一番速くレコードにアクセスできるコマンドです。ところが、クライアント-サーバ環境では、多数のリクエストが発生し、システム全体がスローダウンする恐れがあります。ネットワークが混雑している場合は、さらに深刻な問題になるかもしれません。

結論として、NEXT RECORD コマンドを While または For のどちらでループしても実質的には変わらないということ、GOTO SELECTED RECORD コマンドは許容できるレベルであるということ、そして GOTO RECORD はパフォーマンスを低下させる危険があるということが分かりました。

Example 2: Creating a selection of records

ここでは 3 件のレコードを作成します。

最初のコードでは、単純なループで CREATE RECORD および SAVE RECORD コマンドを繰り返し実行しています：

```
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
```

リクエストログは次のようになりました：

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

SAVE RECORD を実行すると、セレクションがカレントレコード 1 件になります。したがって、次に作成されたレコードがカレントレコードになるときには前のカレントレコードをアンロードする必要があります。クライアントは合計 288,264 バイトのデータを送信しました。

次のコードでは、ARRAY TO SELECTION コマンドでレコードを作成しています：

```
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]fTe
xt1;$at2;[Data]fText2;$at3;[Data]fText3)
```

リクエストログは次のようになりました：

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

リクエスト数がわずか 2 件であり、クライアントは 288,203 バイトしか送信しなかった点に注目することができます。多数のレコードをまとめて作成する場合には ARRAY TO SELECTION コマンドを使用するのがベストです。もちろん、クライアントとサーバの両方に充分のメモリがなくてはなりません。

Example 3: Modifying a selection records

複数のレコードをバッチ処理で更新する方法には、いろいろなパターンが考えられます。

最初の方法は、セレクションの中を単純なループで巡って、SAVE RECORD と NEXT RECORD を繰り返すというものです。ここでは 8 件のレコードを更新します：

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

リクエストログは次のようになりました：

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

最初の 3 件のリクエストは ALL RECORDS のコールです。その後、レコードを更新するたびにロード、セーブ、アンロードが繰り返されています。レコードはサーバからクライアントへ送られ、クライアントで更新され、それからサーバへ送り返されます。合計 27 件のリクエストが発生しました。

次の例では APPLY TO SELECTION でレコードを更新しています：

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

リクエストログは次のようになりました：

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

8 件のレコードを更新するために 32 件のリクエストが発生しました。ここでもレコードが丸々サーバからクライアントへ送信されていることが分かります。単純なループと比較して増えたリクエストは、リレートのチェック、APPLY TO SELECTION では必ず使用される LockedSet システムセットのチェック、そして最後のレコードアンロードです。

ストアドプロシージャを使用して APPLY TO SELECTION を実行するという方法もあります：

```
$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

このようにリクエスト数は 2 件ですが、Lockedset をチェックするメカニズムを別に用意する必要があります。CPU 専有率にも注意する必要があります。サーバの活動が高くなる代償として他のクライアントが影響を受けるかもしれません。

ARRAY TO SELECTION が SAVE RECORD よりも高速であることはすでに分かっているので、配列を使用してレコードを更新する方法を最後に試してみましょう。
もちろん、レコードはサーバからクライアントへ送信する必要がありますが、SELECTION TO ARRAY の良いところは、該当するフィールド以外のフィールドは送信されないという点です。
ARRAY TO SELECTION を実行すると、既存のレコードについては、パラメータで渡されたフィールドが更新されます。レコードが存在しない場合、新規レコードが作成されます。

```
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

まず ALL RECORDS でセレクションが作られ、最初のレコードがロードされます。SELECTION TO ARRAY で配列が作成され、クライアントで更新された後、サーバに送り返されます。ちなみに SELECTION TO ARRAY でカレントレコードがアンロードされることもリクエストログから分かります。リクエスト数は 9 件に減り、転送されるデータの量も少なく済みました。データベースの構造が許すようであれば、これはおすすめのテクニックです。

Example 4: Transactions

今度は、トランザクションの中で SELECTION TO ARRAY を使用し、レコードを作成するという例です。コードは次のようなものになります：

```
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

リクエストログから分かるように、トランザクションで送信されたのはわずか 4 件のリクエストに過ぎませんでした。よほど大量のトランザクションを短時間で実行するのでない限り、トランザクションがネットワークに負担をかけるということはないはずです。

Example 5: Sorting a selection

次の例では、セレクションのレコードを並び替えるというものです。ここでは、発生するリクエストの数を調べることが目的なので、並び替えるレコードの数は問題ではありません。

```
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

ALL RECORDS はおなじみの 4 リクエストを発行しているのに対し、ORDER BY は 5 件のリクエストを送信していることが分かります。ひとつ目のリクエストがソートの実行、続くふたつのリクエストが新しいセレクションの受信です。最初のレコードがロードされるのに備えて以前のレコードはアンロードされています。

Example 6: Searching a selection

次に簡単なクエリを実行してみましょう。

```
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

9 件のリクエストが発行され、サーバは合計で 994 バイトを受信し、314 バイトを返しました。バージョン 6.5 以降、同じ操作を配列でできるようになりました。QUERY WITH ARRAY を使用した場合、結果は次のようになりました：

```
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

4 件のリクエストが発行され、サーバは合計で 83 バイトを受信し、348 バイトを返しました。このようなクエリを実行する場合、QUERY WITH ARRAY コマンドの使用を検討してみる価値があるといえるでしょう。

Example 7: Transferring variables in C/S mode

次の例は、GET PROCESS VARIABLE コマンドを使用し、4D Server から変数を受け取るというものです。変数にはテキスト配列と文字列配列を使用します。それぞれ要素数は同じで、内容的にも同じ 20 バイトのデータを含んでおり、サーバ側でインタープロセス配列として定義されています。

```
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

GET PROCESS VARIABLE を実行するたびに、1 件のリクエストが発生することが分かりますが、同じデータであるにも関わらず、違うバイト数が返されました。要するに、文字列配列はテキスト配列よりも余分に場所を取るということです。この点は、4D Client と 4D Server の間でデータを交わす際に影響を及ぼします。例えば、ストアドプロシージャを作る場合には、パラメータとして配列を渡すことができないので、必然的に配列を BLOB に変換して渡すことになるわけですが、この BLOB のサイズは、元の配列が文字列配列、あるいはテキスト配列であったかによってサイズが変わってきます。BLOB が大きければ、それだけネットワーク通信に時間がかかることになります。

Example 8: Transferring variables in C/S mode

最後の例は、セマフォを使用する際には定番となっているコードです：

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

上記のコードでは、ストアドプロシージャを作り、セマフォを定義する余裕を 4D Server に与えるために DELAY PROCESS を実行しています。このストアドプロシージャは完了するまでに 1 秒以上を要するので、このような方法を使用することができます。P_Idle メソッドは次のようなものです：

```
$a:=Semaphore("Wait ")
For ($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

何と 19000 件以上のセマフォリクエストがサーバに送られたことが分かります。サーバは、毎回、そのようなリクエストに対応しなくてはなりません。CPU とネットワークに負担がかかり、パフォーマンスが低下する可能性があります。

別の方法として、IDLE コマンドの代わりに DELAY PROCESS(0)を使用することにしましょう。

```
$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

19102 件だったリクエストの数が 12709 件まで減りました。

以下は秒毎に 4D Client から 4D Server へ送られたリクエストの詳細です：

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

このように、ストアドプロシージャが完了するまでに要した時間は同じ 10 秒ですが、IDLE コマンドのほうが多くのリクエストを送っていることが分かります。

IDLE コマンドは、アイドルを強制するコマンドというわけではありません。アイドルができるかを調べるコマンドです。アイドルは、そのプロセスに少なくとも 1 ティックが費やされたときに実行できます。例えば、セマフォの確認が実行され、次いで IDLE のリクエストが送られたとします。プロセスはまだ 1 ティックを使い切っていないので、引き続きセマフォの確認が行われ、そして IDLE コマンドが実行されます。結局、1 ティックが経過するまでこの過程が繰り返されます。これに対し、DELAY PROCESS コマンドを実行した場合はアイドルが強制され、処理が次のプロセスに渡された後、カレントプロセスに戻されます。DELAY PROCESS (1)が 1 ティックの間、プロセスを遅らせてアイドルを強制するのに対し、DELAY PROCESS (0)はプロセスを遅らせることなくアイドルを強制するという点が異なっています。

Summary

いろいろなテストコードを実行することによって、どれだけの回数、どのようなリクエストがサーバに送られるのかを調べることができました。カレントレコードのロードとアンロードの

タイミングについても考察することができました。こうして得た情報に基づき、クライアント-サーバを最適化する方法、つまり 4D Server と 4D Client の通信を減らすことによってサーバの CPU 使用を抑え、余力を他の処理に回したり、あるいは単純に処理を高速化できるような方法を考えることができます。ネットワークにかかる負荷も少なくすることができます。

とはいえ、リクエストの数を減らすことがすぐさまパフォーマンスとなって返ってくるわけではありません。テストした限りにおいては、スピードの改善は明白でしたが、実際のクライアント-サーバは複雑なシステムであり、毎回、期待した結果が必ず得られるというわけではないからです。