CA Gen Asynchroneous Communucations Guide

The Best Kept Secret

Rony Van Hove, Sr. Principal Consultant

24 May 2009

Transforming IT Management

CA Gen has been in the market since the late 1980's. It has proven its usefulness and productivity to large corporations around the world and CA recognized this through the years. The use of CA Gen's technologies for CA's Mainframe 2.0 strategy highlights the importance and enduring factor of CA Gen.

New features and capabilities are introduced by each subsequent release of CA Gen, some of the features are not well known to the development community. One such feature is Asynchroneous server calls. In today's pressure for better Service Levels and efficiency, the asynchroneous feature may prove a significant tool to achieve better response times, increase service levels and productivity. This Guide explains how you can benefit from it.

CA Gen Asynchronous communications 4
Synchronous vs. Asynchronous 4
Asynchronous Communication Support in CA Gen 6
Statements: 6
USE ASYNC 6
target_server_pstep 6
POLL | NOTIFY EVENT | NO RESPONSE 6
viewname async_request 7
event_name 7
PSTEP | GLOBAL 7
Scenarios: 9
Asynchronous Processing – Classic way 9
Asynchronous Processing with Event Actions 10
Asynchronous Fire-and-Forget Processing 11
Special Asynchronous Call Cases: Simple calls 11
The GLOBAL-scope technique 11
Program Flow of a Search Procedure: 12
Procedure Step body of OPEN window event of SEARCH Client Procedure 13
Event Actions to Ignore or New request. 14
Event Action 'SHOW_RESULTS: 15
Other Event Actions 16
Program Flow of the CLIENT_MENU procedure 16
Procedure Body of OPEN event of Client_Menu Procedure 17
The POLLEVENT Event Action: 17
View the Results or Ignore Events 18
RESULTS command in the SEARCH CLIENT PROCEDURE 18
When to poll? 19
Special Asynchronous Call Cases: Multiple Calls 19
Conclusions: 21

History

Ver.	Status	Who	Date	Changes, Actions
1.0	Final	Rony Van Hove	27/9/2003	Initial version
1.1	Final	Rony Van Hove	14/2/2004	Translated from Dutch to English.
1.2	Final	Rony Van Hove	17/8/2004	Re-branding from Advantage to AllFusion and corrected a few errors.
1.6	Final	Rony Van Hove	14/05/2006	Minor corrections
1.7	Final	Rony Van Hove	22/02/2007	Revised
2.0	Final	Rony Van Hove	24/05/2009	Revised

CA Gen Asynchronous communications

This Guide describes the asynchronous communication feature of CA Gen. This functionality offers new possibilities to improve performance for critical applications. The Guide details best practices how this feature could be best used for specific situations.

The supplied documentation with the software doesn't provide practical examples or guidelines how this could be implemented. This Guide addresses this topic in detail.

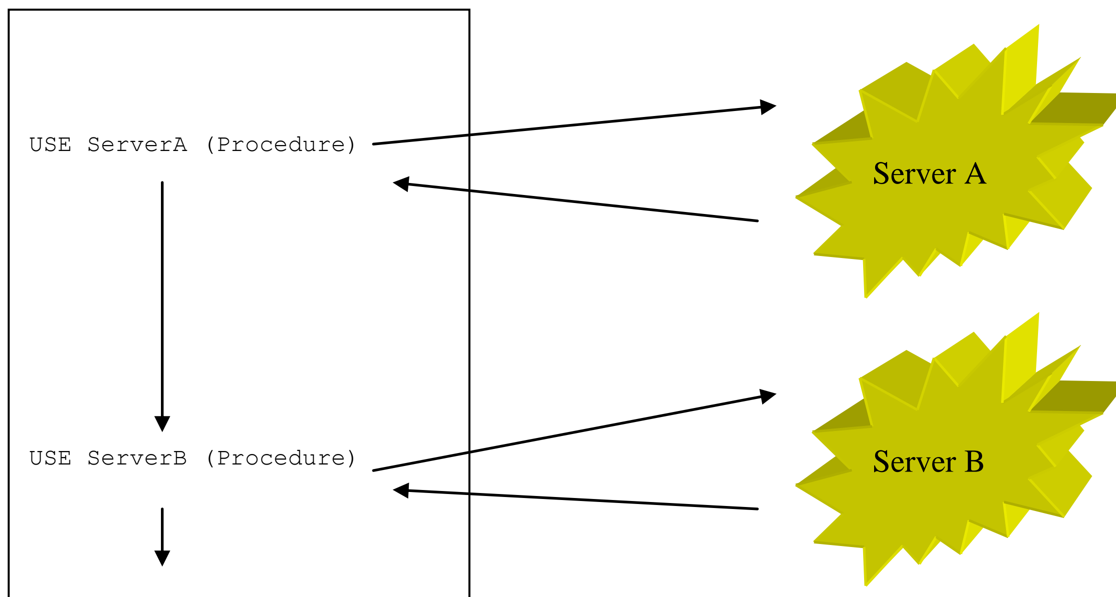
Synchronous vs. Asynchronous

You could only use synchronous server calls in pre-COOL:Gen 5.0 releases; i.e. the server was called and the client had to wait until the response came back from the server. All client-processing was stopped. This technique is also called often '***Blocking Calls***'. This way of calling a procedure step is the most common even today.

The synchronous behavior is in the most cases fine because the sequence of actions or processes required the output of the preceding process.

However occasionally there are situations that this behavior is not best suited and could cause serious performance complains from end-users; e.g. a complex server procedure would a lot of DB2 actions could take several seconds before it completes, this has the negative effect that the end-user has to wait couple of seconds before he can do something.

Example:



If you have sequences of server procedure calls then the blocking effect could have an adverse effect on the end-user experience of the application. The user is often disgruntled and he feels 'not being' serviced properly because he has to wait too long before an answer comes.

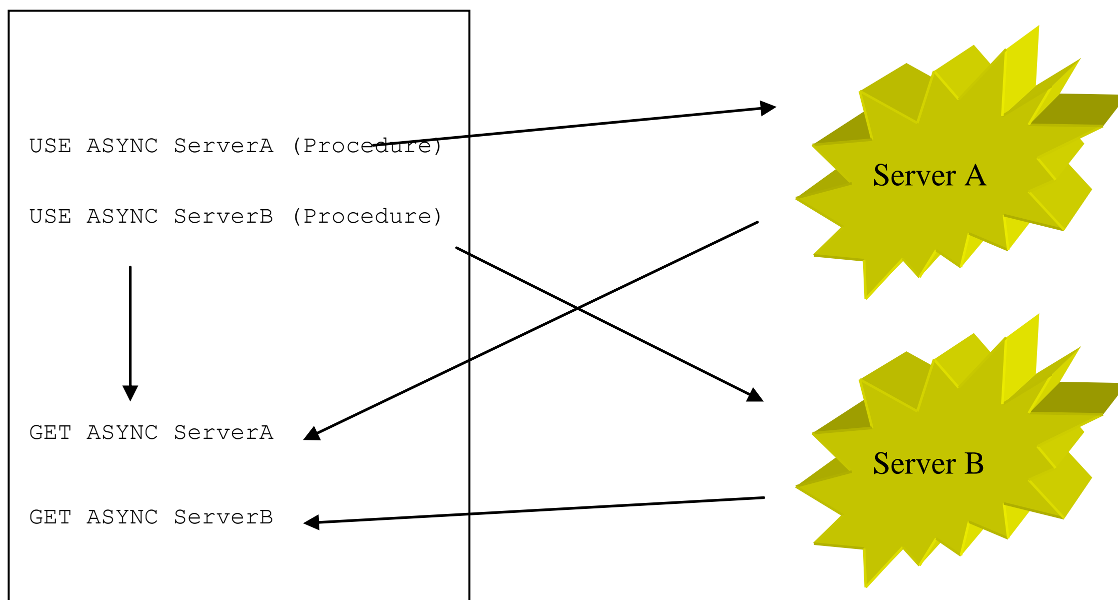
To resolve this problem CA Gen introduced asynchronous communication calls. This was first introduced with COOL:Gen 5.0 for a selected set of middlewares. This feature was further extended to other middlewares in 6.0 release of CA Gen to ensure that all users can enjoy this feature.

To enable this feature some extra statements have been introduced: Asynchronous USE statements. The new statements make asynchronous programming remarkable easy and a breeze in day to day use. Asynchronous programming techniques are often complex and prone to errors and hard to debug. CA Gen made this easy and quick by hiding the complexity of all the thread handling etc… Every CA Gen developer is able to implement in less than a day a asynchronous communications.

Existing applications can be enhanced by ***a few statements*** to benefit from asynchronous communications. No major redesign is required.

An existing client procedure can be tailored for asynchronous communications in a half day or less.

Example



The asynchronous USE statements allow you to execute multiple server transactions in parallel (e.g. A and B) without waiting before one of them is finished. The procedure logic after the last async USE statement will continue. There is ***no wait*** for response from the server procedures.

It is possible to capture the response or even to ignore the response (=***fire-and-forget***) of the server(s).

This concept is perfect when you have search requests that may take considerable long time to fulfill; the end-user is not being blocked by the complex search requests and he can continue with other stuff in the application until he gets a notification that the response came back. This notification event may fill up a listbox or write an entry in a table that can be consulted afterwards.

No server procedure medications are required at all!

> ***NOTE::***
>
> If you use TCP/IP as your preferred middleware then you must regenerate and relink the server transactions with CA Gen 6.x, r7.x
>
> COOL: Gen 5.x doesn't support TCP/IP asynchronous calls.
>
> Existing pre COOL:Gen 5.x (4.1A, Composer 4, Composer 3 etc) server transactions need to be regenerated and relinked with at least 5.x in order to take advantage of this feature. We do recommend when you have to regenerate to ensure that you are using CA Gen 7.6.

Asynchronous Communication Support in CA Gen

CA Gen introduces several new statements for this feature:

USE ASYNC	Call server asynchronous.
GET ASYNC	Get server response from an asynchronous call.
CHECK ASYNC	Check the status of an asynchronous call.
IGNORE ASYNC	Ignore the response of an asynchronous call.

Statements:

USE ASYNC
This is the asynchronous server call variant of 'Procedure Step USE'-statement. The statement will call the server transaction and it will pass the control back to the client immediately without waiting for the response.
Unlike Procedure Step USE statement, this statement allows you to verify the return code of your server transaction request.

The complete syntax of the statement:

```
+--USE ASYNC target_server_pstep
|  IDENTIFIED BY: viewname async_request
|  RESPONSE HANDLING: POLL | NOTIFY EVENT | NO RESPONSE
|  RESPONSE EVENT:  event_name
|  RESPONSE SCOPE: PSTEP | GLOBAL
|  WHICH IMPORTS…
+--WHEN request accepted
+--WHEN request not accepted
+--
```

target_server_pstep: name of the server procedure step.

POLL | NOTIFY EVENT | NO RESPONSE: Type of asynchronous server call:

`POLL`	The response of the server will later be processed.
`NOTIFY EVENT`	If the response is available from the server then an notify event <*event_name*> will be triggered and executed.
`NO RESPONSE`	Ignore the response of the server (***fire-and-forget***).

viewname async_request: workset view for the result of the asynchronous call request.

> **TIP:**
> We recommend to use for each new asynchronous server procedure call to use its own separate workset view to store the result. This practice will make inspection of the worksets of several different asynchronous calls afterwards easier and avoids mistakes.

This workset contains the data concerning the request whether it was a successful call or a failure etc.

ID	NUMBER(8)	Unique ID of the call; each server call is assigned with an unique ID. This ID should be used to recover the response, check the status or to ignore the response.
REASON_CODE	CHAR(8)	Reserved, for future use.
LABEL	MIXEDTEXT(128)	Optional field, can be used for a more userfriendly message
ERROR_MESSAGE	MIXEDTEXT(2048)	Runtime error message

> The fields LABEL and ERROR_MESSAGE are defined as '*MIXED Text*' in CA Gen.

If a call to a server procedure fails then there is no standard error dialog box shown. The error is captured by the ASYNC call statement. The actual error message (TIRMxxx message) is stored in the field *ERROR_MESSAGE*.

event_name: name of the event action (only valid for NOTIFY EVENT).

PSTEP | GLOBAL: this indicates the scope of the asynchronous call:

PSTEP: if you use a PSTEP-scope then the answer of the server transaction is only accessible and valid as long as the Procedure Step is running. As soon as the procedure step is finished due an ESCAPE or a normal ending and you didn't process the response then the response is lost.

GLOBAL: the response is never lost after you have left the procedure step (or flowing to another procedure step etc). The response can be checked and recovered at any place in a procedure step.

NOTICE:

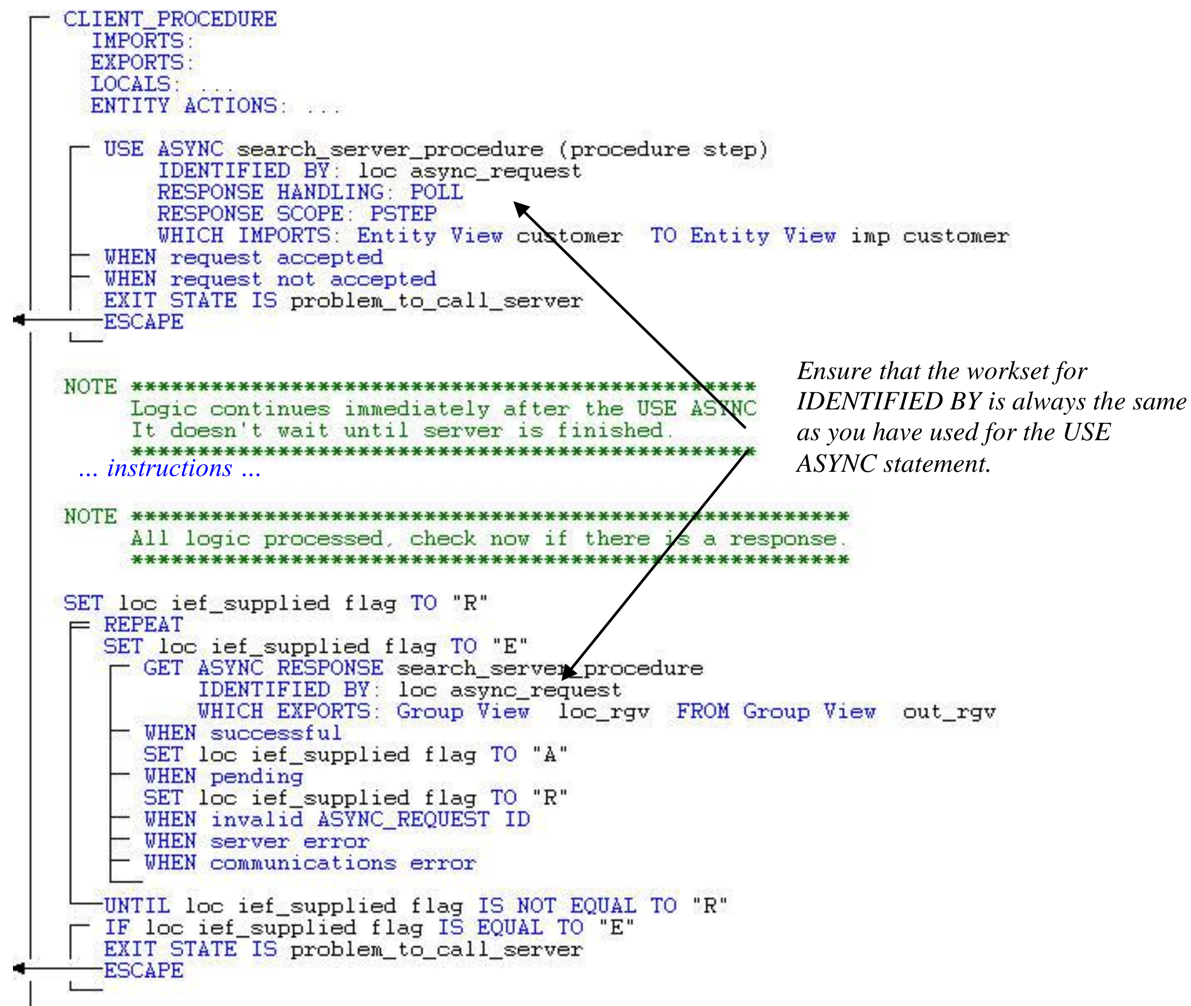
If you don’t use the GLOBAL scope carefully then it may cause memory problems after long non-stop usage of the client part (i.e. no exiting of the client). Each request requires approx. up to 32K of storage to store the results of a server procedure step. If the client never checks this response and thus never freeing the storage of the response then after a long time there will be no more memory left for new calls or other things. You have basically created a memory leak by not checking the response.

The only solution for this problem is to exit the application and restart it again.

More info in section ***Special Asynchronous Call Cases.***

Scenarios:

Asynchronous Processing – Classic way



The USE ASYNC is called in the beginning of the procedure. The control is passed back immediately to the procedure once the server transaction is started. The client procedure is now able to process other statements. At the end of the procedure, you have to verify the answer(s) from the server transaction(s).

The REPEAT-UNTIL construction is required because we cannot predict when the server transaction is finished. If the server is not finished then the GET_ASYNC will give a PENDING as response.

Additionally you can add special processing in case of a communication error, server error or an invalid ID.

Basically the REPEAT-UNTIL continues polls when this is finished.

The scope for this case should be PSTEP i.e. the response will only be processed in this procedure.

Asynchronous Processing with Event Actions

```
NOTIFY_EXAMPLE
  IMPORTS:
  EXPORTS:
  LOCALS: ...
  ENTITY ACTIONS: ...

SET loc async_request id TO 0
  USE ASYNC search_server_procedure (procedure step)
      IDENTIFIED BY: loc async_request
      RESPONSE HANDLING: POLL
      RESPONSE SCOPE: PSTEP
      WHICH IMPORTS: Entity View customer  TO Entity View imp customer
  WHEN request accepted
  WHEN request not accepted
  EXIT STATE IS problem_to_call_server
  ESCAPE

EVENT ACTION data_found_event
  GET ASYNC RESPONSE search_server_procedure
      IDENTIFIED BY: loc async_request
      WHICH EXPORTS: Group View  loc_rgv  FROM Group View  out_rgv
  WHEN successful
  SET loc async_request id TO 0

  NOTE ***************************
       Process response now
       ***************************

  WHEN pending
  WHEN invalid ASYNC_REQUEST ID
  EXIT STATE IS problem_to_call_server
  ESCAPE
  WHEN server error
  EXIT STATE IS problem_to_call_server
  ESCAPE
  WHEN communications error
  EXIT STATE IS problem_to_call_server
  ESCAPE
```

An asynchronous call is initiated and when a response is available then an event is kicked-off.

The response type for this call should be ***NOTIFY EVENT*** *(see above example in NOTIFY_EXAMPLE procedure)*. The event action should always in the same procedure step. This event action will contain the instructions to process the response.

Because this NOTIFY EVENT requires an event action and these events are always local to the procedure, the scope of the call is always PSTEP.

The workset view for *async_request* must turn off the flag '*Initialize on every entry*' if this is a local work view.

Asynchronous Fire-and-Forget Processing

```
FIRE_AND_FORGET_EXAMPLE
  IMPORTS:
  EXPORTS:
  LOCALS: ...
  ENTITY ACTIONS: ...

SET loc async_request id TO 0
USE ASYNC search_server_procedure (procedure step)
    IDENTIFIED BY: loc async_request
    RESPONSE HANDLING: NO RESPONSE
    WHICH IMPORTS: Entity View customer  TO Entity View imp customer
WHEN request accepted
WHEN request not accepted
EXIT STATE IS problem_to_call_server
ESCAPE
```

The Fire-and-Forget construction uses the handling type ***NO RESPONSE***. You never have to check the response of the server.

This is the perfect construction to submit transactions which are ‘semi-batch’.

Special Asynchronous Call Cases: Simple calls

There could be situations where asynchronous calls are executed, but their response should be handled somewhere else in the application.

An example: an end-user is launching a heavy search request in a window and since it is a heavy-duty server transaction, it will be started in asynchronous mode.

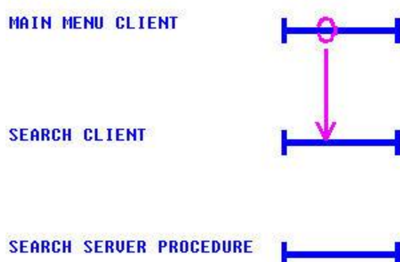
The user may now continue with other things while the search transaction is in progress in the background (server).

Once the response is available from the server, the results may not be directly viewable because the procedure step that started the call may no longer be active.

CA Gen wouldn’t be CA Gen if there is no solution for this type of cases. CA Gen has a special instruction to cope with these situations: the ***GLOBAL***-scope.

The GLOBAL-scope technique

Dialog Flow:



From the main menu client you can flow to a window for lookups. This lookup procedure will be able to kick-off asynchronous calls for complex search patterns.

The end-user may when necessary flow back to the main menu and perform some other procedures while the search in the background is in progress.

Use the following programming technique:

Program Flow of a Search Procedure:

Start

Another request in process?

Yes

No

Ask user to interrupt the previous request or wait for that response

Cancel the previous request?

Yes

No

Ignore the previous request

Wait until a response is available.

Enter new search pattern

Call the server asynchronously

Show results

Because the client procedure works event-driven, we will code the different sections of the diagram in event actions:

Event ***SHOW_RESULTS***: will be called via a TIREVENT. This event will process the response and show the results.

Event ***ENTER_NEW_SEARCH_PATTERN***: will be called via a TIREVENT. This event will display a dialog box to enter the search pattern. If the user clicks OK then it will start the search transaction.

Both events could happen at any time, so therefore we have to define them as user-defined events via TIREVENT.

Example code fragements:

Procedure Step body of OPEN window event of SEARCH Client Procedure
The procedure step will check the COMMAND: SEARCH or RESULTS. Why the command RESULTS is required will be explained later.

If the command SEARCH is used then the procedure step will check whether an outstanding search request exists. If this is the case then the user will be asked whether to ignore the previous request or wait for the results of that request.

```
SEARCH_CLIENT
  IMPORTS: ...
  EXPORTS: ...
  LOCALS:
    Work View   loc ief_supplied
      command
      flag
  ENTITY ACTIONS:

MOVE imp_search customer  TO out_search customer
MOVE imp async_request  TO out async_request

  CASE OF COMMAND
  CASE results

  NOTE ****************************************
       Responds checking was outside this
       procedure step, so get the results now.
       ****************************************

  SET loc ief_supplied command TO "RESULTS"
  USE tirevent
      WHICH IMPORTS: Work View   loc ief_supplied  TO Work View   imp ief_supplied

  CASE search

  NOTE **********************************
       Check for any outstanding request.
       **********************************

    IF out async_request id IS NOT EQUAL TO 0

    NOTE **********************************
         Yes, still an outstanding request.
         **********************************
    OPEN Dialog Box OUTSTANDING_SEARCH_REQUEST

    ELSE

    NOTE ******************************
         No outstanding request.
         Display the search dialog box.
         ******************************
    OPEN Dialog Box ENTER_NEW_SEARCH_PATTERN
```

Event Actions to Ignore or New request.
There are two possible answer for the above question:

Yes – ignore the previous request

No – wait for response.

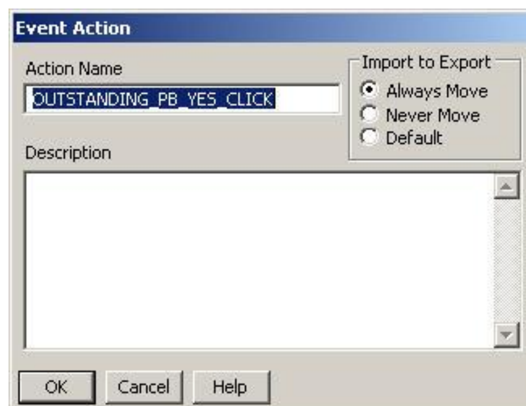
This means we need two additional click events:

```
EVENT ACTION outstanding_pb_yes_click
CLOSE Dialog Box OUTSTANDING_SEARCH_REQUEST

NOTE ************************************
     Outstanding request may be ignored.
     ************************************

  IGNORE ASYNC RESPONSE
      IDENTIFIED BY: out async_request
  WHEN invalid ASYNC_REQUEST ID

SET out async_request id TO 0

EVENT ACTION outstanding_pb_no_click
CLOSE Dialog Box OUTSTANDING_SEARCH_REQUEST

NOTE *****************************************
     Don't ignore the outstanding request.
     So wait until server responds to previous
     request.
     *****************************************

SET loc ief_supplied flag TO "R"
  REPEAT
     CHECK ASYNC RESPONSE
         IDENTIFIED BY: out async_request
     WHEN available
     SET loc ief_supplied flag TO "N"
     WHEN pending
     WHEN invalid ASYNC_REQUEST ID
  UNTIL loc ief_supplied flag IS EQUAL TO "N"
SET loc ief_supplied command TO "RESULTS"
USE tirevent
    WHICH IMPORTS: Work View   loc ief_supplied  TO Work View   imp ief_supplied
```

IGNORE gives the possibility to ignore the response.

Code to wait until response is available from previous request

Attention: you should enable for both click-events the flag '***Always***' for '***Import to Exprt***':



If the user clicks ***No*** then the client will wait until a response is available. Once this response is available the user-event 'SHOW_RESULTS' is activated via TIREVENT.

NOTE:

Be aware that we always set the asynchronous ID to 0 after the response has been processed. While this is not strictly required, we highly recommend to do this. This allows us afterwards whether an outstanding request is still in progress or not.

If the user wants ignore it (***YES***) then the client should ignore the outstanding request and the asynchronous ID is set to 0.

NOTE:

IGNORE ASYNC RESPONSE doesn't interrupt or cancel the server transaction, it will only discard the response message from memory as soon as the response comes from the server.

The asynchronous call ID identifies the server request. If this is the wrong ID then it may discard a server's response that you'll need, therefore it is important that you keep attention which ID you are ignoring. Using separate worksets for this view is a good technique.

Event Action 'SHOW_RESULTS:

The event action *SHOW_RESULTS* will show the results of a request:

```
EVENT ACTION show_results
NOTE ******************************************
     Get the results and show it on this window.
     ******************************************

   GET ASYNC RESPONSE search_server_procedure
       IDENTIFIED BY: out async_request
       WHICH EXPORTS: Group View  out_rgv  FROM Group View  out_rgv
   WHEN successful
   WHEN pending
   WHEN invalid ASYNC_REQUEST ID
   WHEN server error
   EXIT STATE IS problem_to_call_server
   SET out async_request id TO 0
   ESCAPE
   WHEN communications error
   EXIT STATE IS problem_to_call_server
   SET out async_request id TO 0
   ESCAPE

SET out async_request id TO 0
```

Ensure that the flag 'Always' is set for ***Import to Export*** on the Event-Action properties.

Other Event Actions
If there is no outstanding request then it will fire off the user-event 'ENTER_NEW_SEARCH_PATTERN'. This will show a dialogbox to enter your search pattern.

If the user clicks 'Search' on that dialog box then it will start the asynchronous call:

```
EVENT ACTION enter_new_se_pb_cancel_click
CLOSE Dialog Box ENTER_NEW_SEARCH_PATTERN
NOTE ****************************************
     Cancel : don't search.
     ****************************************

EVENT ACTION enter_new_se_pb_search_click
CLOSE Dialog Box ENTER_NEW_SEARCH_PATTERN

NOTE ****************************************
     Send my request to server asynchronously.
     ****************************************

   USE ASYNC search_server_procedure (procedure step)
       IDENTIFIED BY: out async_request
       RESPONSE HANDLING: NOTIFY EVENT
       RESPONSE EVENT: show_results
       RESPONSE SCOPE: GLOBAL
       WHICH IMPORTS: Entity View imp_search customer  TO Entity View imp customer
   WHEN request accepted
   WHEN request not accepted
   EXIT STATE IS problem_to_call_server
   SET out async_request id TO 0
   ESCAPE
```

*Please note that RESPONSE SCOPE is set to **GLOBAL** instead of PSTEP.*

If there is a response from the request while the procedure step is still active then the event 'SHOW_RESULTS' will be activated.

Program Flow of the CLIENT_MENU procedure

If the clients decides not to wait for the response and flows back to the client procedure then the NOTIFY EVENT will not be active anymore when the response should arrive.
You lose the automatic event activation because the client procedure for search is no longer active and the runtime cannot 'see' anymore the event.
You still have to verify the response in one way or another. If you don't do this then you will create memory leaks and would eventually run low in memory and cause crashes.

There are various methods and techniques to circumvent this.

However, one of the best mechanisms is explained here:

In our example, we are going to use a polling mechanism in order to simulate the response notify event of the search client procedure.

This can be materialized by usea of user-events in the client menu procedure.

For this purpose, you will need one view of the workset type ***async_request***. We ensure that the flag '***Initialize on every entry***' is turned off for this view.

The polling happens on basis of an user-event, this technique allows us to launch the polling at any given time in the procedure step. However, the polling request needs to be done explicitly.

You may also use the Timer Control event that was originally supplied with Composer 4/COOL:Gen 4.1A and later releases. This has the advantage that you don't have to launch explicitly the polling event.

Procedure Body of OPEN event of Client_Menu Procedure

```
MAIN_MENU_CLIENT
  IMPORTS:
  EXPORTS:
  LOCALS: ...
  ENTITY ACTIONS:

NOTE ***********************************
     Check whether polling is required.
     ***********************************

IF loc_server async_request id IS NOT EQUAL TO 0
SET loc ief_supplied command TO "POLLEVENT"
USE tirevent
    WHICH IMPORTS: Work View   loc ief_supplied  TO Work View   imp ief_supplied
```

In this sample, we check whether we have to launch the polling event. The request ID is basically the asynchronous ID. If the value is 0 then no outstanding request is pending.

If different from 0 then there is a polling required (you may now understand also why we set always ID to 0 after a request's response is handled in the search client procedure)

The POLLEVENT Event Action:

```
EVENT ACTION poll_response
  IF loc_server async_request id IS NOT EQUAL TO 0
    CHECK ASYNC RESPONSE
        IDENTIFIED BY: loc_server async_request
    WHEN available

    NOTE ********************************
         There is a response available.
         ********************************

    OPEN Dialog Box RESPONSE
    WHEN pending
    WHEN invalid ASYNC_REQUEST ID
```

In this example, the POLLEVENT will check whether a request is still outstanding (this is a good practice to handle unforeseen circumstances).

Afterwards a CHECK ASYNC RESPONSE will verify whether there is already a response available from the request. If it is available then a small notification dialog box is shown to tell the user that a response is available. The user will now have the opportunity to go back to the search client window in order to view the results:

The actual retrieval of the response happens in our search client procedure.

NOTE:
The CHECK ASYNC RESPONSE will never remove the response from its memory and therefore a GET ASYNC or IGNORE ASYNC is still needed.

View the Results or Ignore Events
These two events handle the answer of the user whether to view it or not.

```
EVENT ACTION response_pb_ignore_click
CLOSE Dialog Box RESPONSE
 IGNORE ASYNC RESPONSE
     IDENTIFIED BY: loc_server async_request
 WHEN invalid ASYNC_REQUEST ID

SET loc_server async_request id TO 0

EVENT ACTION response_pb_view_click
CLOSE Dialog Box RESPONSE
COMMAND IS results
EXIT STATE IS go_to_search
```

If the user clicks the ‘NO’ button then the response will be put in the garbage bin via the IGNORE ASYNC RESPONSE.

If the user wants to view it (‘YES’) then the command ***RESULTS*** is set and we flow to the search client procedure.

RESULTS command in the SEARCH CLIENT PROCEDURE
If the user responds positively then it flows to the search client procedure. The command will now be checked for RESULTS:

```
SEARCH_CLIENT
  IMPORTS: ...
  EXPORTS: ...
  LOCALS:
    Work View   loc ief_supplied
      command
      flag
  ENTITY ACTIONS:

MOVE imp_search customer  TO out_search customer
MOVE imp async_request  TO out async_request

  CASE OF COMMAND
  CASE results

  NOTE ****************************************
       Responds checking was outside this
       procedure step, so get the results now.
       ****************************************

  SET loc ief_supplied command TO "RESULTS"
  USE tirevent
     WHICH IMPORTS: Work View   loc ief_supplied  TO Work View   imp ief_supplied

  CASE search

  NOTE **********************************
       Check for any outstanding request.
       **********************************

     IF out async_request id IS NOT EQUAL TO 0

     NOTE **********************************
          Yes, still an outstanding request.
          **********************************
     OPEN Dialog Box OUTSTANDING_SEARCH_REQUEST

     ELSE

     NOTE ******************************
          No outstanding request.
          Display the search dialog box.
          ******************************
     OPEN Dialog Box ENTER_NEW_SEARCH_PATTERN
```

RESULTS command check.

The user-event ***SHOW_RESULTS*** will be activated.

When to poll?

A crucial question is when to execute polling event?

The general rule we recommend is to perform a polling event every time you return from a flow. This can happen in a OPEN window event or in the procedure body.

Potentially, you may consider to perform a polling event in each click event. However this would be potentially an overkill and a serious overhead if there are many events.

Special Asynchronous Call Cases: Multiple Calls

Let us now increase the level of complexity; we want to launch multiple search requests... This require some serious management of the asynchronous responses (thus ID etc). Each request has its own unique ID and this ID need to be used in order to differentiate from the others.

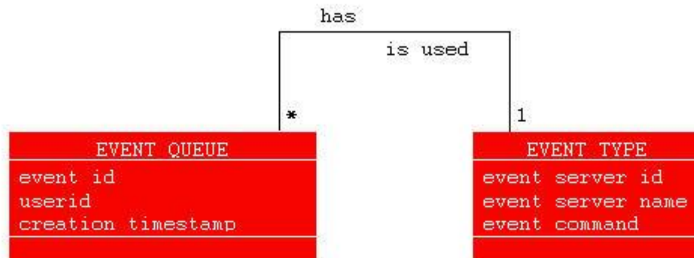
The ID doesn’t tell you anything which server transaction it belongs to. So, we also need a mechanism to associate an ID with a server transaction.

Another complexity is that we also have to check whether there are any outstanding requests etc.

Multiple requests to various server transactions require asynchronous call management. It is simpler when you allow multiple requests for the same server transaction, however you still need some sort of management of these calls.

A possible implementation is to use an event table (in a local database via ODBC) or via a RGV.

The below example gives such an event table structure:



The ***Event Type*** table contains all possible server procedure calls who can be called in asynchronous mode with GLOBAL scope.

The table ***Event Queue*** contains per user which outstanding server calls are in progress.

For each new request a new occurrence is written in the ***Event Queue*** table for the specified user. This table should be verified during a poll event (or polling action block).

Additionally, it would be possible to extend the table ***EVENT TYPE*** with an extra field 'COMMAND'. If a need to be processed then it can use the corresponding command of that event type. A TIREVENT needs to be called in order that the correct user-event is executed.

If you use a RGV then you can opt for a simplified structure; define one workset with two attributes: ID and COMMAND. Each server has its own specific command.

The RGV should be defined in the procedure or passed to the possible procedures that may contain polling events.

It is also important that if you use many procedures that need to check for any outstanding requests then it is wiser to use action blocks in your polling event… just to avoid that you have much coding work.

An example polling event (usage of a RGV):

```
EVENT ACTION polling_voor_antwoord
  IF loc_events IS NOT EMPTY
  NOTE ********************************
       Er zijn openstaande asynchrone aanvragen
       ********************************
    FOR SUBSCRIPT OF loc_events  FROM 1  TO LAST OF loc_events  BY 1
      IF loc_event_item async_server id IS NOT EQUAL TO 0
      SET loc_req async_request id TO loc_event_item async_server id
        CHECK ASYNC RESPONSE
            IDENTIFIED BY: loc_req async_request
        WHEN available
        SET loc ief_supplied command TO loc_event_item async_server command
        USE tirevent
            WHICH IMPORTS: Work View   loc ief_supplied  TO Work View   imp ief_supplied
        WHEN pending
        WHEN invalid ASYNC_REQUEST ID
```

In this case the RGV is checked for any outstanding requests. If there is an outstanding request (<>0) then it checks whether there is a response available from that request. If this is the case then it will queue up an event via its corresponding command and TIREVENT.

For each call to an asynchronous server you should set a command and the ID for the RGV:

After an answer is processed the RGV needs to be 'corrected' (to avoid overflows):

```
IF SUBSCRIPT OF loc_events IS LESS THAN LAST OF loc_events
SET loc_event_item async_server id TO 0
ELSE
SET SUBSCRIPT OF loc_events  TO SUBSCRIPT OF loc_events - 1
  WHILE SUBSCRIPT OF loc_events IS GREATER THAN 0
    IF loc_event_item async_server id IS NOT EQUAL TO 0
    SET LAST OF loc_events  TO SUBSCRIPT OF loc_events
<-- ESCAPE
    ELSE
    SET SUBSCRIPT OF loc_events  TO SUBSCRIPT OF loc_events - 1
```

Whether you chose for table or RGV approach you always have to foresee something to monitor the outstanding requests.

Conclusions:
For simple cases you can consider GLOBAL scope and it would mean that you have to track the asynchronous ID and carry it along to the procedure steps in order to be able to process it

afterwards. You can carry over the ID with dialog flow view matching or by USE view matchings.

For multiple requests of multiple servers you should have a proper request management system in place (preferable via a template). If you don't do this then you will create havoc in memory management (creating memory leaks) due the unprocessed responses etc.

The user should also be aware that each request requires up to 32K to process the answer.

If you have 100 outstanding calls then the application will claim 100x32K = 3MB additional memory in the worst case.