MiniM InterConnect

Eugene Karataev
support@minimdb.com
http://www.minimdb.com/tools/interconnect.html

November 26, 2016

Contents

Chapter 1

Introduction

MiniM Interconnect is a TCP/IP layer to connect two jobs of different MUMPS servers in a client-server mode.

Interconnect works in a client-server mode, when one process on one side calls second process on a other side and both processes are on different MUMPS servers. MUMPS servers can be MiniM, Cache or GT.M in any pairs, any versions and on any supported by vendors operating systems.

Both sides are implemented as MUMPS routines, with need vendor-specific code to use custom InterConnect protocol over TCP/IP network connection.

MiniM InterConnect was based on the MiniM Server Connect protocol and after porting one to other MUMPS systems is a usable tool to connect two MUMPS servers. MUMPS servers can have owned protocols to implement server-server connection. GT.M implements OMI, Cache implements ECP, and MiniM now does not has custom distributed connection protocol. MiniM InterConnect is a vendor-independent protocol to connect different MUMPS servers.

This tool can be ported to any other MUMPS system with support of TCP/IP devices.

To setup MiniM InterConnect administrator imports with compilation need side of protocol - server, client or both and make settings of TCP/IP port where server side should wait incoming connections. This port and address of this server need to be used in connecion code on the client side. By default, MiniM InterConnect uses ports 5000 for MiniM, 5001 for Cache and 5002 for GT.M. Different ports are used to allow servers to work on the same computer.

MiniM InterConnect can connect not only different MUMPS servers but also servers of one vendor, and with different versions: MiniM to MiniM, GT.M to GT.M or Cache to Cache. For example, connect Cache with ECP

and Cache without ECP support.

Application Programming Interface of MiniM InterConnect consists not of direct global access, it is client-server protocol, where client side evaluates expressions on the server, executes line of commands and gets data by different modes. So developers can use any server-side specific functions and commands and call subroutines on the server in full-functional client-server mode.

On the server side MiniM InterConnect runs one MUMPS job to each connection and client side can be connected to different servers at the same time. Administrators should know about process usage on the server - each connection use one job and plus one job is used by the MiniM InterConnect daemon.

Server side of MiniM InterConnect is the same as used by MiniM Client Tools, Cache Tools and GT.M Tools from www.minimdb.com, so if this tools already in use, server side of MiniM InterConnect already is ready to use.

MiniM InterConnect have not special requirements to operating system, processor or bitness of processor, becouse was written on the MUMPS.

Chapter 2

Installation

2.1 Cache'

Server-side part of MiniM InterConnect for Cache is a TCP/IP daemon. Is is special background job which wait incoming TCP/IP connection and run child process for each connection. This part was written in MUMPS and contains in the %srv routine.

MiniM InterConnect works with 8-bit Cache instances only and has not unicode version.

To import this routine go to the Cache portal, select the %SYS database and import the srv.rsa file. Or run the Cache Studio and import this routine in XML format from the %srv.xml file. This files contains the same routine and differs only by formats.

Routine can be loaded using Cache Portal, Cache Studio, or from terminal

```
USER>zn "%sys"
%SYS>d $System.OBJ.Load("path/to/%srv.xml","C")
```

This commands will import routine with compilation.

Or use the %RI utility in the %SYS namespace:

```
USER>zn "%sys"

%SYS>d ^%RI

Input routines from Sequential
Device: path/to/srv.rsa
Parameters? "RS" =>
```

```
File written by OLD %RO on 04 Dec 2015
   4:20 PM with description:
Cache for Windows NT^^Export 1 routines
   from database USER^~Format=Cache.S~

( All Select Enter List Quit )

Routine Input Option: all Routines

If a selected routine has the same name as
  one already on file,
shall it replace the one on file? No => Yes
Recompile? Yes => Yes
Display Syntax Errors? Yes => Yes

^ indicates routines which will replace those
  now on file.
@ indicates routines which have been [re]compiled.
- indicates routines which have not been filed.


%srv.INT^@

1 routine processed.
```

In all places where %RI asks what to do, enter first letter of your answer and press Enter, utility completes answer.

After this routine is ready to work. To run daemon open terminal or telnet and execute in any database code to run daemon:

```
USER>d ^%srv
Cache TCP server ^%srv.
Cache TCP server ^%srv has been run.
```

By default, daemon waits incoming TCP/IP connection on the port 5001. To change this value, enter the port number (for example, 5055) into settings global:

```
USER>s ^%SRV("port")=5055
```

This value is used on the daemon starting. So stop the daemon by running:

```
USER>d stop^%srv
```

And run again:

```
USER>d ^%srv
Cache TCP server ^%srv.
Cache TCP server ^%srv has been run.
```

If this %srv daemon active, MiniM InterConnect can be run and connected to this InterSystems Cache instance.

To run MiniM InterConnect daemon automatically on Cache start, edit special %ZSTART routine. Label SYSTEM in this routine Cache executes each time when starts. So, add after this label code to run MiniM InterConnect daemon like the following:

```
SYSTEM ;
    ; Cache starting
    d ^%srv
    ...
```

To see more information about %ZSTART routine, see InterSystems Cache documentation.

While daemon works, he writes diagnostic messages to builtin Cache log file cconsole.log.

This daemon already in use by Cache Tools, so if Cache Tools has been already run, server side of MiniM InterConnect is ready to run too.

To install client side of MiniM InterConnect for Cache, import routine %cli from the Cache subdirectory of MiniM InterConnect installation archive.

2.2 GT.M

Server-side part of MiniM InterConnect for GT.M is a TCP/IP daemon written in MUMPS. This additional job opens TCP/IP port, waits incoming connections and runs job to each one to service requests. All code of this daemon is in one routine %srv.

MiniM InterConnect works with 8-bit GT.M instances only and has not unicode version.

Select subdirectory for GT.M where unpacked zip with MiniM InterConnect. There is file %srv.rou with server side routine. First of all this routine must be imported into GT.M by using builtin routine import utility

```
GTM>d ^%RI
```

Enter there real full name of the file srv.rou.
After importing compile them by call

```
GTM>zcompile "_srv.m"
```

Underscores are used by GT.M in file names instead of the percent symbol.
And link routine by call

```
GTM>zlink "_srv.m"
```

After this steps routine %srv is ready to be used.
This daemon uses by default TCP/IP port number 5002. To change this, write into global port number, for example to use port number 6003 execute:

```
GTM>s ^%SRV("port")=6003
```

To stop %srv daemon manually or from other procass call

```
GTM>d stop^%srv
```

This code fire stop indicator and returns immadiately. Stop indicator periodically checked by %srv daemon and daemon will stop.
To start %srv daemon manually or from other process call

```
GTM>d ^%srv
```

This code runs MUMPS job for daemon and returs immediately.
To prevent manually enter this code on each start of GT.M server, add running of this code into the GT.M run script. See file gtmstart at point where this script runs GT.CM server. This may looks like this:

```
echo "Starting GT.CM (${service}, ${id})..."
if [ ! -d $logdir/$service ]
then
  $echo "logging directory (${logdir}/${service}) ...
  mkdir $logdir/$service
fi
nohup $gtm_dist/gtcm_run -service $service -id ${id ...
  >> $logdir/${service}/session.log 2>&1 < /dev/null &
```

```
    if [ $? != 0 ]; then
      $echo "The GT.CM server (${service}) failed to start."
    fi
    sleep 1
    done < $gtm_dist/gtcm_slist
```

And after running GT.CM add running of %srv daemon by adding code

```
$gtm_dist/mumps -r ^%srv
```

And gtmstart script may looks like this:

```
    echo "Starting GT.CM (${service}, ${id})..."
    if [ ! -d $logdir/$service ]
    then
      $echo "logging directory (${logdir}/${service}) ...
      mkdir $logdir/$service
    fi
    nohup $gtm_dist/gtcm_run -service $service -id ${id ...
      >> $logdir/${service}/session.log 2>&1 < /dev/null &

    if [ $? != 0 ]; then
      $echo "The GT.CM server (${service}) failed to start."
    fi
    sleep 1

    # Run tcp/ip listener ^%srv
    $gtm_dist/mumps -r ^%srv

    done < $gtm_dist/gtcm_slist
  fi

else
  exit
fi
```

On running of GT.M start sequence we may see additional diagnostic message:

```
Starting the GT.CM server(s).
Starting GT.CM (omi, 42)...
GT.M TCP server ^%srv.
GT.M server ^%srv is running.
```

%srv daemon additionally writes internal diagnostic messages into system log and messages can be viewed in syslog in the /var/log subdirectory like the following line:

```
Jul  9 13:01:44 debian GTM-MUMPS[2137]:
   GT.M server ^%srv is running.
```

This daemon already in use by GT.M Tools, so if GT.M Tools has been already run, server side of MiniM InterConnect is ready to run too.

To install client side of MiniM InterConnect for GT.M, import routine %cli from the GT.M subdirectory of MiniM InterConnect installation archive.

2.3 MiniM

MiniM instances by default contains the %srv routine and by default this routine updates on upgrade. Bydefault %srv daemon for MiniM InterConnect is used by all MiniM GUI Tools and by MWA. So default MiniM installation is ready to run server-side MiniM InterConnect.

By default the %srv daemon runs from the autostart.m script and this daemon uses by default TCP/IP port 5000. To change this port, write into global port number, for example to use port number 6003 execute:

```
%SYS>s ^%SRV("port")=6003
```

To run this daemon if one was not running call

```
%SYS>d ^%srv
```

and to stop daemon call

```
%SYS>d stop^%srv
```

To install client side of MiniM InterConnect for MiniM, import routine %cli from the MiniM subdirectory of MiniM InterConnect installation archive.

2.4 Examples

Examples are additional code samples how MiniM InterConnect can be used, and does not used by MiniM InterConnect server- or client-side routines. This examples can be imported and removed any time and are included only for developers.

Chapter 3

Application Interface

3.1 Variables

MiniM InterConnect API uses only one variable - %cli and this variable should not be newed.

MiniM InterConnect API supports several connections to different servers in the same time and each connection identifies by one value - internal identificator of connection.

This identificator is not zero for established connection and zero if opening of connection fails. Identificator is an index value inside of %cli variable and all assosiated with connection data are stores inside this subnodes. Different versions of MiniM InterConnect API can have different internal structures.

To identify MiniM InterConnect connection use only one value:

```
s connection=$$Open^%cli(params...)
...
d Proc^%cli(connection,params...)
...
d Close^%cli(connection)
```

3.2 Open

The Open function creates new connection to server side and accepts TCP/IP server name (or TCP/IP address), TCP/IP port where server side runs and database name if connects to MiniM or Cache server.

```
$$Open^%cli(server,port,database)
```

Return value is 0 if connection cannot be established or non-zero value otherwise.

After creating connection this client job owns one more opened TCP/IP device, so do not close all devices by argumentless CLOSE command.

```
s server="192.168.2.12"
s port=5000
s database="user"
s connection=$$Open^%cli(server,port,database)
i 'connection d  q
. w $$Error^%cli(connection),!
...
d Close^%cli(connection)
```

If server side is GT.M, the database parameter is ignored, and for MiniM or Cache server side swithes to specified database.

3.3 Close

The Close subroutine terminates connection with the server side daemon and removes internal data associated with this connection.

```
d Close^%cli(connection)
```

After this call don't use this connection value.

3.4 Error

Function Error returns string with last error occured on last call or empty string on no error. This string contains values of $zerror for MiniM and Cache and value derived of $zstatus value with indicator of error text and possible error place.

Short example to call GT.M server and show

```
USER>s connection=$$Open^%cli("localhost",5002,"")

USER>s name=$$Read^%cli("$zv")

<UNDEFINED> :Read+1^%cli: *%cli("$zv","dev")
```

```
USER>s name=$$Read^%cli(connection,"$zv")

USER>w name
GT.M V6.2-002A Linux x86
USER>w $$Error^%cli(connection)

USER>s name=$$Read^%cli(connection,"undefined")

USER>w $$Error^%cli(connection)
%GTM-E-UNDEF, Undefined local variable: undefined at eval^%srv
USER>s value=$$Read^%cli(connection,"any illegal syntax")

USER>w $$Error^%cli(connection)
%GTM-E-INDEXTRACHARS, Indirection string contains extra
trailing characters at eval^%srv
USER>

USER>d Close^%cli(connection)
```

Real string with error is dependent of the server side server and server version. String with last error occured on the server is stored inside the %cli variable for each connection.

3.5 Read

Function Read returns value of exaluated on the server side MUMPS expression. Expression can be specified as any valid MUMPS expression including variable names, builtin functions, operators and other language elements.

```
$$Read^%cli(connection,expression)
```

Return value is stored as it was evaluated on the server, including all nonprintable characters. If server returns list strictures or any special strings, function Read return entire this value.

Example:

```
%SYS>s value=$$Read^%cli(connection,"$c(13,10)")

%SYS>zzdump value

0000: 0D 0A
```

3.6 Write

Function Write assignes value to variable on the server side.

```
Write^%cli(connection,varname,varvalue)
```

In the varname can be used any local, global, or assignable system variables or indirection expression valid for the SET command. String with variable name must have correct MUMPS syntax if contains indices.

```
%SYS>d Write^%cli(connection,"vvv","123")

%SYS>w $$Read^%cli(connection,"vvv")
123
```

And can be recommended before writing to variable with indices write the name first and write value with indirection.

```
%SYS>d Write^%cli(connection,"tmp","cmpnd(""ind"")")

%SYS>d Write^%cli(connection,"@tmp","789789")

%SYS>w $$Read^%cli(connection,"cmpnd(""ind"")")
789789
```

3.7 Kill

Function Kill kills specified variable name or indirectly specified variable name on the server side.

```
Kill^%cli(connection,varname)
```

Here in the varname can be used local, global variables, and system or structured system variables for which server supports KILL command.

3.8 Execute

Function Execute calls server side to execute line of commands as an argument of the XECUTE command. Here can be directly and indirectly specified commands.

```
Execute^%cli(connection,commands)
```

Argument "commands" can contain any valid XECUTE argument supported by the server. Be careful with dots - they are not supported by MUMPS and with QUIT command with an argument - in most cases it is an error dataflow.

Example:

```
%SYS>d Execute^%cli(connection,"s rrr=456")

%SYS>w $$Read^%cli(connection,"rrr")
456
```

3.9 OnGroupRead

Subroutine OnGroupRead assignes label with need to be called when server uses special data transfer in "GroupRead" mode. When server code executes and calls

```
d wo^%srv(datastring)
```

then on the client fires event OnGroupRead with entire value passed from the server (here in code it is datastring).

Label need to be specified with routine name becouse this code must be work while routine %cli is current routine.

Label must be passed without dollar signes, quotes and other symbols.

Label must acceps arguments:

```
OnGroupRead(connect,value)
```

Label does not quit any value and called as subroutine using DO command.

Here connect have current connection descriptor value and value contains data was passed from the server.

Example:

```
...
d OnGroupRead^%cli(connect,"OnGroupRead^CLIEXAM4")
...
s commands="s i="""" f  s i=$o(^ROUTINE(i)) q:i="""""
   d wo^%srv($lb(i,$zd($g(^ROUTINE(i,0)),3)))"
```

```
 d Execute^%cli(connect,commands)
 ...
 ; unpack name and date and print
OnGroupRead(connect,value)
 w "Routine name: ",$lg(value,1),", date change: ",$lg(value,2),!
 q
```

Here is routine from example set CLIEXAM4. Code calls OnGroupRead subroutine and pass label and routine name to be event handler. Before calling this label code of %cli restores current device which was before calling %cli routine.

Be careful that this event handler executes before then code of Execute returns control. Event handler OnGroupRead fires to each call of wo^%srv.

Internal data transfer protocol of this event delivers data of string passed from the server to the client without waiting of confirmation, so data transfer code can work as fast as possible. This data transfer mode is mostly recommended to fill string lists, string grids, transfer big texts, and other transfer of homogenous set of data strings.

This event can occures at any server call except ExecuteOutput - there all data stream transfers as is, without delimiting to separate data strings.

To remove this event handler, pass empty string:

```
 d OnGroupRead^%cli(connect,"")
```

If no event handler was assigned, no any label on the client will be called even if server calls wo^%srv, this data will be lost on the client.

This event can be called in any context except ExecuteOutput subroutine was called.

3.10 OnOutput

Subroutine OnOutput assignes event handler which fires when server writes data to current device while works subroutine ExecuteOutput. All data stream passed to event handler with undefined length until zero bytes reached. Zero byte will be send by server side after last commands executes. So this data transfer mode is not recommended for data passing when server can pass zero byte as part of data.

```
 d OnOutput^%cli(connection,label)
```

Label need to be specified with routine name becouse this code must be work while routine %cli is current routine.

Label must be passed without dollar signes, quotes and other symbols.

Label must accept arguments:

```
OnOutput(connect,data)
```

Label does not quit any value and called as subroutine using DO command.

Here connect contains current connection descriptor and data contains next part of data passed from the server. The length of this data is undefined and passed as received from the TCP/IP socket. If server side calls special write arguments such as "w !" or "w ?", client side cannot recognise this write forms and accepts real bytes passed from the server to current device. In depending of server software, for example, command "w !" can pass to the socket the sequence

```
 $C(10)
```

or

```
 $C(13,10)
```

Example:

```
 ...
 d OnOutput^%cli(connect,"OnOutput^CLIEXAM2")
 ...
OnOutput(connect,str)
 w str
 q
```

Here all output received from the server simply writes out to current device.

Before calling this label code of %cli restores current device which was before calling %cli routine.

The OnOutput catching mode can be recommended to receive ordinal output to current device made by compilers and ordinal CHUI software without using escape sequences.

To remove this event handler, pass empty string:

```
 d OnOutput^%cli(connect,"")
```

If no event handler was assigned, no any label on the client will be called even if server writes to current device, this data will be lost on the client.

This event can be called only in the ExecuteOutput context.

3.11 OnCallback

Subroutine OnCallback assignes event handler for calling client back from the server while server code calls $$cb^%srv(datastring).

When server side calls $$cb^%srv(datastring), it waits while client side receives datastring and make answer. Server side receives answer from the client and continues execution.

```
 d OnCallback^%cli(connection,label)
```

Label need to be specified with routine name becouse this code must be work while routine %cli is current routine.

Label must be passed without dollar signes, quotes and other symbols.

Label must accept arguments:

```
OnCallback(connect,datastring)
```

Label requires QUIT value which will be passed to the server side back and this value will receive $$cb^%srv(datastring) and this label on the client side calls as user function $$.

To remove this event handler, pass empty string:

```
 d OnCallback^%cli(connect,"")
```

If no event handler was assigned, no any label on the client will be called, data passed from the server will be lost on the client and server side will receive empty string.

Example:

```
 ...
 d OnCallback^%cli(connect,"OnCallback^CLIEXAM5")
 ; call routine which call client back
 s value=$$Read^%cli(connect,"$$callback^example5()")
 ...
OnCallback(connect,command)
 n value,answer
 ; in this contex we can call MiniM
 ; in example 5 we ignore Command value
 s value=$$Read^%cli(connect,"$zt($p($h,"","",2))")
 ; compose callback answer
 s answer=" time: "_value
 ; trace execution order
```

```
 n saveio s saveio=$io u $p
 w "Client on command """,command,""" return: ",answer,!
 u saveio
 q answer
```

Routine example5:

```
callback()
 n value="date: "_$zd($h,2)
 s value=value_$$cb^%srv("get string")
 q value
```

Here client side calls server to read value of $$callbackˆexample5() or call function. While function executes, it calls client back using $$cbˆ%srv("get string"). This passes "get string" to client and waits responce.

On the client fires event handler OnCallback(connect,command) and there client reads back the server

```
 s value=$$Read^%cli(connect,"$zt($p($h,"","",2))")
```

Server receives Read command, evaluates expression and return value of $zt(...).

After this client side make answer from the OnCallback handler

```
 s answer=" time: "_value
```

and return it

```
 q answer
```

After this server side receives responce to $$cbˆ%srv("get string") and continue execution.

This data transfer mode can be recommended when server need to send command to client or ask additional information. This mode works with saving stack levels.

This event can be called in any context except ExecuteOutput subroutine was called.

3.12 ExecuteOutput

Subroutine ExecuteOutput executes line of MUMPS commands on the server like the Execute subroutine, but intercepts all output made by the server side to current device until zero byte is reached.

It is special context when client does not recognize any control sequence and all output pass to event handler of OnOutput. In this mode don't call GroupRead or Callback functionality.

Subroutine Execute instead of ExecuteOutput recognizes control sequences and while Execute works, all OnGroupRead and OnCallback events will fire.

Chapter 4

Uninstallation

4.1 Cache

To remove MiniM Interconnect daemon stop it first:

```
%SYS>d stop^%zsrv
```

Next remove server side routine %srv.
To remove client side of MiniM InterConnect remove routine %cli.

4.2 GT.M

To remove MiniM Interconnect daemon stop it first:

```
GTM>d stop^%srv
```

Next remove server side routine %srv.
To remove client side of MiniM InterConnect remove routine %cli.

4.3 MiniM

To remove MiniM Interconnect daemon stop it first:

```
%SYS>d stop^%srv
```

Next check that MiniM InterConnect daemon is not in the start sequence in autostart.m script file.
And next routine %srv can be removed.

And you need to know that this daemon in use by all MiniM Client Tools, by MWA and by all other tools where used MiniM Server Connect.

Routine %srv by default replaces by upgrade process if you install new version of MiniM over existing.

To remove client side of MiniM InterConnect remove routine %cli.