Article
Alexey Maslov · Nov 17, 2016 11m read

ECP and Process Management API

The technology of load balancing between several servers with relatively low capacity has been a standard feature of Caché for quite a while. It is based on the distributed cache technology called ECP (Enterprise Cache Protocol). ECP provides a host of possibilities for horizontal scaling of an application, and yet keeping the project budget fairly low. Another apparent advantage of ECP network is the possibility to conceal its architecture in the depths of Caché configuration so that applications developed for the traditional (vertical) architecture can be fairly easily migrated to a horizontal ECP environment. The ease of this process is so mesmerizing, that you start wishing it was always this way. For instance, everybody is used to having a possibility to control Caché processes: the $Job system variable and associated classes/functions work magic in skilful hands. Stop, but now processes can end up being on different Caché servers…

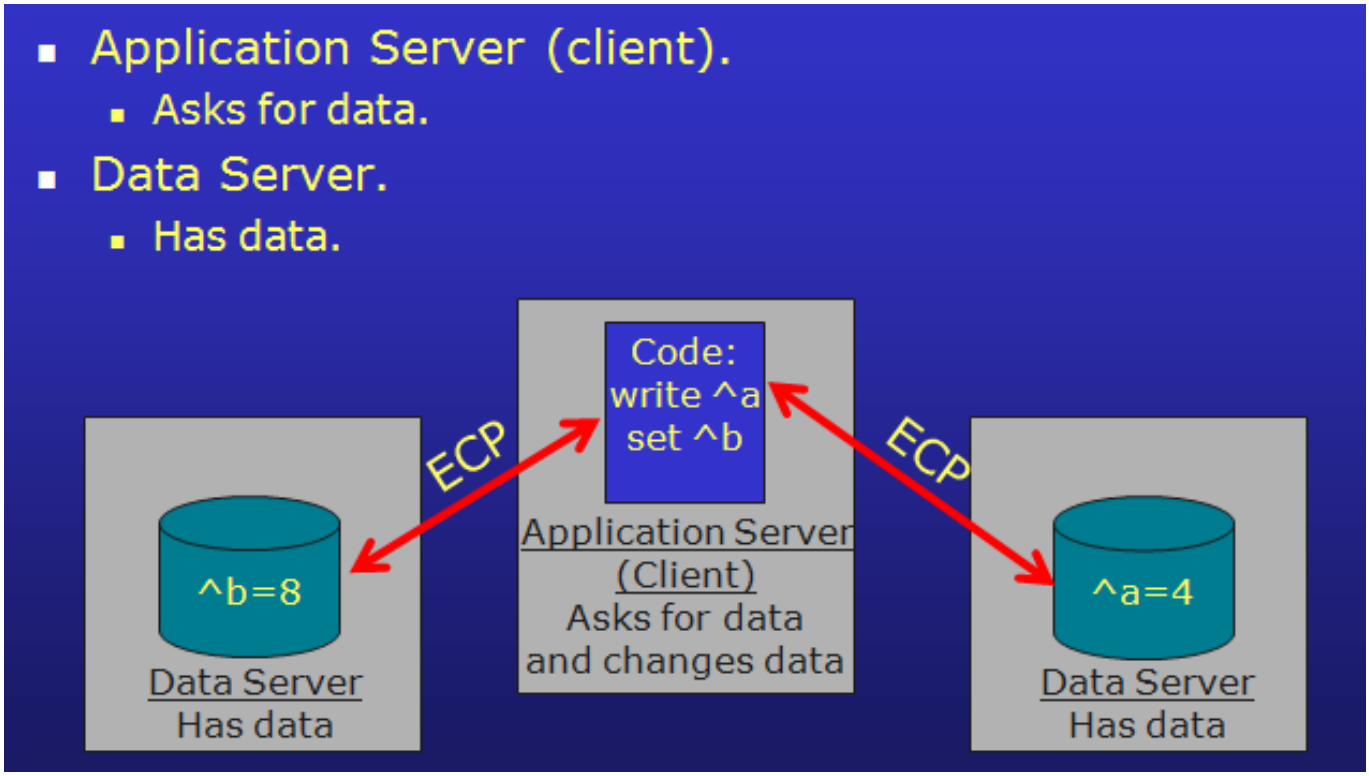
This article is about how to gain as much transparency in controlling processes in ECP environment as in traditional (non ECP) one.

The ABC of ECP

Before we dive into this topic, let ' s recall the standard ECP terms.

ECP, or enterprise cache protocol, is one of the key technologies that allow data servers and application servers to interact. It works on top of TCP/IP, which ensures reliable transportation of data packets. ECP is the property of InterSystems.

Caché instances involved in ECP networking are usually considered to be either Application or Data servers. The principal scheme of their interaction is illustrated below.



Key components of ECP

If ECP concepts are quite new for you, I encourage you to go through the best introduction I've ever met, named ECP Magic. Understanding the basics will be enough for further reading.

If you wish to learn ECP concepts in depth, I'd like to recommend to start with Murray Oldfield's excellent article Data Platforms and Performance - Part 7 ECP for performance, scalability and availability.

ECP from a Programmer ' s Perspective

" Normal " programs work in namespaces, which means that you won ' t need to change constructs like ^|"^^c:\intersystems\cache\mgr\qmsperf"|QNaz to ^|"^MyDataSrv^c:\intersystems\cache\mgr\qmsperf"|QNaz. All of these details are packed into definitions of the remote data server and remote database inside Caché configuration. Even if you eventually need to address globals in namespaces that are different from the current one, the syntax of such queries (^|"qmsperf"|QNaz) will remain the same.

The semantics of working with global data also remains virtually unchanged – it ' s just that we rarely think about it while working with local databases. Let ' s list the key rules:

• All operations are divided into synchronous (all read function: $Get, $Order, etc., as well as Lock and $Increment) and asynchronous (data saving: Set, Kill, etc.). Further execution of the program does not require the completion of asynchronous operations. However, things are different for synchronous ones. In case of ECP, they may require additional requests to the data server, if a data block is not found in the local cache.

• Synchronous operations do not wait for asynchronous operations (initiated by the same application server) to finish.

• The Lock command waits for data writing operations (started by the previous lock owner) to finish.

• Expiry of the Lock timeout period doesn ' t automatically mean that the lock is owned by someone else.

And here are some very peculiar details:

• Assignment

```
set i=$Increment(^a)
```

can be more expensive than its functionally similar analogue:

```
lock +^a set (i,^a)=^a+1 lock -^a
```

The thing is that the $Increment function is always run on the data server, so waiting for packages to complete their journey back and forth is inevitable, and Lock causes such an effect only when its requester and its current owner reside on different application servers.

• You need to process <NETWORK> errors. They occur when an application server cannot restore a lost ECP connection within *Time to wait for recovery* period (1200 seconds by default). The correct way to handle them is to roll back the already started transaction and try again.

• (Effective till Caché v2016.1) Strings longer than a half of a block are not cached on application servers. In reality, this threshold is a bit lower – around 3900 bytes for 8 KB blocks. This decision was made by the developers to keep cache clean of BLOBs and CLOBs: such data is usually written once and is rarely read afterwards. Unfortunately, this decision negatively affected the processing of bitmap indexes that are, as a rule, long strings, too. If you use them, you will either need to reduce the chunk size or increase the block size; the optimal size can

be determined only after testing.

ECP and Process Management

Having refreshed our memory on basic ECP concepts, let's proceed to the main topic of this article. Let's look at process management from a broad perspective. From that of the real needs of application developers and system tools that are there for them " right out of the box " – see below.

The main process management needs

Function	Without ECP	With ECP
Start of new background processes	job $job, $zchild, $zparent	The Job command works in the netwo Process IDs are unique only within a
Monitoring of process activity	$data(^$job(pid))	No access to the process table of and
Getting a list of processes	$order(^$job(pid)) $zjob(pid)	See above.
Access to the properties of other processes	Class %SYS.ProcessQuery	See above.
Termination of another process	Class SYS.Process	It is impossible to terminate a process

These challenges were addressed by the development of a process management API that was implemented as the Util.Proc class.

To keep you interested in reading along, let me show you a couple of simple examples using the API.

Util.Proc API use cases

• Show a list of processes, including the name of the namespace and the name of the user of the health information system (HIS), marking your own process with an asterisk ("*"):

```
 set cnt=0
 for {
     set proc=##class(Util.Proc).NextProc(proc,.sc) quit:proc=""||'sc
// next process
     write proc_$select(##class(Util.Proc).ProcIsMy(proc):"*",1:"")
// marking itself with «*»
     write " namespace: "
     write ##class(Util.Proc).GetProcProp(proc,"NameSpace")
// process property: current namespace
     write " user: "
     write ##class(Util.Proc).GetProcVar(proc,$name(qARM("User"))),!
// process variable: user's name
     set cnt=cnt+1
 }
 write "Total: "_cnt_" processes."
```

• Remove a process different from the current one, if it ' s running under the same user's name (for excluding the possibility of duplicate sign-ins):

```
if '##class(Util.Proc).ProcIsMy(proc),
    ##class(Util.Proc).GetProcVar(proc,$name(qARM("User")))=$name(qARM("User")) {
  set res=##class(Util.Proc).KillProc(proc)
}
```

Addressing Processes in the Network

When working on the API, I had to select a method of addressing processes in an ECP network which achieve the following:

• unique addresses in the network,

• possibility to directly use an address with minimal transformations,

• easy-to-read format.

To address a server on the network, you can use its hostname or IP address. Selection of a hostname as an identifier is attractive, but imposes additional requirements for the stability of the name service. Since this is not typically required during Caché configuration, new restrictions would be undesirable. Besides, different operating systems may have a different hostname format, which will seriously complicate the subsequent analysis of the process descriptor. Based on this, I preferred to use an IPv4 address.

In order to identify a Caché instance on a server, you can use its name ("CACHE", "CACHEQMS", etc.) or a superserver TCP port number (1972, 56773, etc.). However, you cannot connect to a Caché instance by its name, so let's select a port.

As the result, a decision was made to use a string in the following format as a descriptor (unique identifier): xx.yy.zz.uu.Port.PID, where

xx.yy.zz.uu is an IPv4 address of a Caché server,

Port – a TCP port of the Caché superserver,

PID – the process ID on the Caché server ($job).

Examples of correct process descriptors:

192.168.11.19.56773.1760 – a process with PID=1760 on a Caché instance with IP=192.168.11.19 and Port=56773.

192.168.11.77.1972.62801 – a process with PID=62801 on a Caché instance with IP=192.168.11.77 and Port=1972.

Methods of the Util.Proc class

As the result, the Util.Proc class was developed, which open methods are listed below. All of them are class methods.

Summary of process management API methods

Method	**Function**
IsECP() As %Boolean	Whether the code is executed i
NextProc(*proc*, ByRef *sc* As %Status) As %String	The next process after the proc
DataProc(*proc*, ByRef *sc* As %Status) As %Integer	If ##class(Util.Proc).DataProc(*p*
GetProcProp(*proc*, *Prop*, ByRef *sc* As %Status) As %String	Get a *Prop* property of the proc queried (see class %SYS.Proce

	Pid, ClientNodeName, UserNam ClientExecutableName.
GetProcVar(*proc*, *var*, ByRef *sc* As %Status) As %String	Get the value of variable *var* of
KillProc(*proc*, ByRef *sc* As %Status) As %String	Terminate the process with des
RunJob(*EntryRef*, *Argv*...) As %List	Start a process on the data serv number of arguments in *Argv*. R data server.
CheckJob(*pid*) As %List	Check whether the process with
CCM(*ClassMethodName*, *Argv*...) As %String	Execute an arbitrary *ClassMeth* server, passing it the necessary result.

By comparing the methods summary with the table *"The main process management needs"* , we will see that we have now managed to satisfy them in the network environment. The CCM() method was added later: in the process or migration of our application (a Wide-Area Health Information System called qMS) to ECP environment, we found out that certain functional blocks should better be run on the data server. The reasons may be quite different:

• Willingness to avoid a one-time transfer of a large amount of data to the application server, which is characteristic of situations when, for instance, a report is generated.

• The need to centrally service a shared resource, such as a message queue with another system (HealthShare in our case).

Let me note that most API methods are intended for working in ECP environment. They are still usable without ECP, but they only send/receive fairly useless process descriptors like 127.0.0.1.Port.pid. The only exceptions are methods intended for working with a data server: RunJob(), CheckJob(), CCM(), as they return/accept the server-side process ID (pid) instead of the process descriptor (proc) itself. Therefore, these methods were made universal from an application developer's point of view: their interface is the same both in ECP environment and outside, although they work quite differently, of course.

A few words about implementation

I needed to select a method of interaction between processes running on different servers. The following alternatives were considered:

• %SYSTEM.Event class.

o Does not work in networks (officially), which means that InterSystems may discontinue the support of its network operation at any moment.

• Full-blown TCP server.

o A generally good idea.

o We need to use an additional TCP port (except for the superserver port), which will inevitably entail additional efforts for installing and configuring more than just standard Caché settings. But nobody wants the additional efforts of such kind...

• Web services.

• %Net.RemoteConnection class. For those who have forgotten: this class supports remote code execution on other servers using the same protocol used by the clients of the %ServiceBindings service. If this service is already being used for connecting clients, no additional settings will be required, and that's exactly our case. Data exchange overheads will be negligible, as they are likely to be smaller than in the case of web services.

Having considered all of the above, I opted for %Net.RemoteConnection. In my opinion, its greatest drawback is that it does not allow to return strings longer than 32 kb, but it has never been much of a problem.

Another interesting challenge that I faced was determining whether the code worked in the network or not. The answer to this question was needed both for internal API needs (in order to generate correct process descriptors), and for writing the IsECP() method that is badly needed by application developers. The reason for such popularity is quite obvious: very few people wanted to refactor parts of their own code related to process interaction in order to use a kind of universal API (although this API had been implemented). It turned out it was a lot easier and more natural to add a branch for ECP. But how to determine in which environment the code is being executed? The following options were considered:

1. The main database of the namespace is remote.

Pro: it's really simple, all you need to do is this:

```
if $piece(##class(%SYS.Namespace).GetGlobalDest(),"^")'=""
// we are in ECP environment
```

Con: this applies to the application server only and excludes the possibility of networking on the data server.

2. No.1 or (the main database of the namespace is mounted by someone remotely). Cons:

This is expensive.

This is unreliable due to the dynamic nature of ECP.

3. No.1 or (the application server connects to the data server using one of its network interfaces).

I went with option 3, since it enables you to relatively quickly find an answer to your question and correctly fill out process descriptors both on the application and data servers. Note that in order to make this check faster, its positive result for each server is registered in the global.

Summary

The successful implementation of a process management API as part of the Wide-Area Health Information System of the Krasnoyarsk Kray demonstrated (at least) the viability of the selected approaches. Using this API, our developers managed to solve a number of important tasks. I will list just a few of them:

- Prevention of duplicate user sign-in's.
- Getting a network-wide list of active users.
- Message exchange between users.
- Starting and monitoring of background processes responsible for lab analyzers.

In conclusion, I would like to thank my colleagues from SP.ARM for helping me to test the code, responding timely to bugs and especially for fixing some of them. Some methods of the Util.Proc class (CCM(), RunJob(), CheckJob()) were made independent from our application and they can be downloaded from the github repository.

Hope this rather long reading was of some use for you... Happy coding!

#Caché #Distributed Data Management #ECP

Source URL:https://community.intersystems.com/post/ecp-and-process-management-api