Summary:

This is a small emulator of Zabbix agent for AS/400 platform. Main purpose: to access AS/400-specific objects (message queues, output queues, subsystems and jobs). It's written on Java using the IBM Toolbox for Java API and library, implemented as Jar-file ready to start (see details below).

Limitations:

- lack of IPv6 (IPv4 only supported);
- no encryption (sorry, no plans);
- only limited subset of standard metrics supported, and even in this case: some of them have a bit different semantics of parameters (see **`proc.num[]`** or **`eventlog[]`** for examples);
- some config file's parameters are recognized, but really ignored (PidFile, EnableRemoteCommand, Alias, AllowRoot, Include, UserParameter and LoadModule);
- configuration parameter ListenIP allows to set only one IP-address (contrary to list in original Zabbix-agent);
- minimum value for the StartAgents parameter is 1 (i.e. active-only mode is not supported);
- during message queue monitoring the integer part only of the message's EventID is transferred to Zabbix-server. It is restriction of Zabbix database schema (it has the integer type for the appropriate attribute). However, it's possible to use a regular expression in the item's key to filter by the full text value of EventID;
- maximum number of ASP's reported is 110, disk units is 174 (restricted by a hard-coded buffer size 16kB).

Requirements:

As this is a Java-program, you need a JRE to use it :-) This Jar-file has been compiled using JDK 1.8, but for running under JRE 1.7. So, you need minimum Java version 1.7. I tested it for monitoring our AS/400 system v 7.1; but it should, probably, work on other versions also.

I tried to have minimum dependencies, but some still exist. You need 2 libraries: IBM Toolbox for Java (`jt400.jar`) and Simple JSON parser (`json-simple-1.1.1.jar`). The first library is included in the AS/400 operating system, or you can download an open-source version (JTOpen) from the sourceforge site (link, you need only **jt400.jar** file from the archive). The second library is a tiny (<25KB) file for parsing JSON text, you can download it here (link, you need the **json-simple-1.1.1.jar** file only).

Really, you need the following files and directories:

- **jt400.jar**;
- **json-simple-1.1.1.jar**;
- **ZabbixAgent.jar** (in fact, Zabbix-agent emulator, this project);

- config file for the agent (default is `zabbix_agentd.conf` in current directory). Example is included, check and modify it for your environment;
- directory where the log file could be written (the only place where the write access is required).

Start of program:

First of all, you need check the config file and modify, at least, the parameters "LogFile" and "Server". You can also modify, if necessary, parameters "ServerActive", "Hostname" and "DebugLevel" ("DebugLevel=4" will produce a lot of debugging information, "DebugLevel=5" additionally will write a debug info from a **collector** thread, see **proc.cpu.util[…]** metric description). Note that monitoring of message queues works only for active mode of Zabbix-agent.

This program could be running on one of two places: either directly on the AS/400 system or on any other host that has JRE and can access AS/400 system via network. In either case you need an AS/400 user profile (to start the program or to connect to system as this user). You need also necessary libraries (`jt400.jar` and `json-simple-1.1.1.jar`) in one of two places: either in the same directory as `ZabbixAgent.jar` file or in the directory for JRE's system libraries (`${JAVA_HOME}/lib/ext/`).

For the first case (running on AS/400 directly) you can start program using the command like the following (entire this command is a single line):

```
SBMJOB CMD(JAVA CLASS('/home/ZABBIX/agentd/ZabbixAgent.jar')
PARM('/home/ZABBIX/agentd/zabbix_agentd.conf')) JOB(ZBXSVC) JOBQ(QSYS/QSYSNOMAX)
USER(ZABBIX)
```

For this command run successfully, you need a directory mentioned (`/home/ZABBIX/agentd/`) where the following files are located: ZabbixAgent.jar, jt400.jar and zabbix_agentd.conf. Also, the link to system's jt400.jar file should be in the "lib" directory of the system JRE (see: http://www-01.ibm.com/support/docview.wss?uid=nas8N1011798, in my case:

```
ln -s /QIBM/ProdData/OS400/jt400/lib/jt400.jar /QIBM/UserData/Java400/ext/jt400.jar
```

).

Config file in this case can have credentials to be commented out. Default credentials are: as400ServerHost=localhost, User=*CURRENT , as400Password=*CURRENT (it's enough for this configuration).

You can also configure AS/400 to start it as a subsystem:

```
CRTLIB LIB(ZABBIX) TEXT('Zabbix stuff')

CRTJOBQ JOBQ(ZABBIX/ZABBIX) TEXT('Zabbix job queue')

CRTSBSD SBSD(ZABBIX/ZABBIX) POOLS((1 *BASE)) TEXT('Zabbix subsystem')

ADDJOBQE SBSD(ZABBIX/ZABBIX) JOBQ(ZABBIX/ZABBIX) MAXACT(*NOMAX)

CRTCLS CLS(ZABBIX/ZABBIX) RUNPTY(35) TEXT('Zabbix class')

ADDRTGE SBSD(ZABBIX/ZABBIX) SEQNBR(9999) CMPVAL(*ANY) PGM(QSYS/QCMD)
CLS(ZABBIX/ZABBIX)

CRTJOBD JOBD(ZABBIX/ZBXSVC) JOBQ(ZABBIX/ZABBIX) TEXT('Zabbix autostart') USER(ZABBIX)
RQSDTA('JAVA CLASS(''/home/ZABBIX/agentd/ZabbixAgent.jar'')
PARM(''/home/ZABBIX/agentd/zabbix_agentd.conf'') JOB(ZBXAGT)')

ADDAJE SBSD(ZABBIX/ZABBIX) JOB(ZBXSVC) JOBD(ZABBIX/ZBXSVC)
```

In this case you can start/stop this agent just starting/stopping this subsystem:

```
STRSBS ZABBIX/ZABBIX
ENDSBS ZABBIX
```

If, by some reason, the subsystem is active but the process is absent, then you can start it using the following command (it is a single line):

```
SBMJOB CMD(JAVA CLASS('/home/ZABBIX/agentd/ZabbixAgent.jar')
PARM('/home/ZABBIX/agentd/zabbix_agentd.conf') JOB(ZBXAGT)) JOBD(ZABBIX/ZBXSVC)
USER(*JOBD)
```

For the second case (network access to AS/400) you can use the command like the following:

```
java -jar [path/to/]ZabbixAgent.jar [[/path/to/]config_file]
```

Reference to conig file is optional, by default the file `zabbix_agentd.conf` in the current directory is used.

In this case you must specify parameters "as400ServerHost" and "as400Password" in config file. You can also specify the "User" parameter (it will have the default value "zabbix" for this configuration).

File listing:

- ZabbixAgent.jar – Zabbix agent emulator;
- zabbix_agentd.conf – example of config file;
- readme.pdf – documentation.

Supported metrics

agent.exit

Causes to program stopped gracefully. Maybe, this metric will be removed in future; however, it is convenient for debugging.

agent.hostname

Agent's hostname according to its config file settings.

agent.ping

Always "1". Could be used to check if the program is running.

agent.version

String with the agent version.

eventlog[name,regexp,severity,source,eventid,maxlines,mode,user]

Messages from the message queue with given name. The first parameter is mandatory (all others are optional): it is the name of message queue; could be specified as fully qualified IFS path name or just as name. In the latter case the fully qualified IFS path name is formed as "`/QSYS.LIB/`" + **name** + "`.MSGQ`". Unlike to standard metric, the **severity** parameter is not a regular expression but a number restricting the minimal level. **regexp**, **source**, **eventid** and **user** are regular expressions, **source** is the name of job generated the message. The **regexp** is compared in case-sensitive manner, but **source**, **eventid** and **user** – in case-insensitive manner.

The appropriate Item should have type "Zabbix agent (active)" and type of information "Log".

For example:

```
eventlog[QSYSOPR,,50,QZ,^CPC1235$,100,skip]
```

Monitor the standard `QSYSOPR` message queue (IFS: "`/QSYS.LIB/QSYSOPR.MSGQ`") for the messages with severity equal or greater than 50 with EventID == "CPC1235" written by jobs with names containing "QZ".

Note 1. Unfortunately, according to standard Zabbix database schema, field for the `EventID` attribute is "Integer". Therefore, it unable to store the full value of original EventID ("CPC1235" in the example above), so the numeric part only transferred to Zabbix ("1235" in this case). Accordingly, this part only will be accessible for the subsequent processing on the server side (for example: used as macro `{ITEM.LOG.EVENTID<1-9>}` or returned value for the trigger function `logeventid()`).

As workaround, by default the full original EventID is inserted to the beginning of message text (using one space character as a separator). This behaviour could be disabled using parameter

```
as400EventIdAsMessagePrefix=0
```

in the config file.

Note 2. Similarly, there is no an additional field in Zabbix database for a **user** metadata (current user name of job that generated this message in a message queue). Therefore, if necessary, this value could be added as a prefix to the message value by setting the following parameter

```
as400UserAsMessagePrefix=1
```

in config file (by default it is disabled). If both (EventId and User) prefixes are used, then the user will be first of them but eventID will follow.

Note 3. We ignore all messages with the "Reply" type. Usually, they are not really useful, often they have no valid EventID; but the most troublesome is inability to seek to the "reply" message. API used has the following note about the **MessageQueue.setUserStartingMessageKey()** method:

```
If the key of a reply message is specified, the message search
begins with the inquiry or sender's copy message associated with
that reply, not the reply message itself.
```

It could cause to infinite loop, so we just ignore such messages.

proc.cpu.util[name,user,type,subsystem,mode,jobnum]

Float: percentage of time the specified job or set of jobs used the CPU during the time defined by **mode** parameter.

Job could be specified by one of two ways:

- using job number (**jobnum**), job name (**name**) and job user name (**user**) triplet. If **jobnum** is specified, **name** and **user** parameters also must be specified (**subsystem** parameter is ignored in this case). This triplet unambiguously identifies a single specific job that should be present in running state.

- if **jobnum** parameter is not specified, then a sum of CPU usage for all running jobs filtered by job name (**name**), user name (**user**) and subsystem (**subsystem**, it is a regular expression) is calculated.

Parameter **type** is used for compatibility only, it could be empty, "total", "used" or "system"; but really ignored.

The **mode** parameter specify a calculation mode, it could be "avg1" (default), "avg5" or "avg15". It defines a time period for statistics calculation (1, 5 or 15 minutes accordingly).

So, for example, `proc.cpu.util[SCPF,QSYS,total,,avg1,000000]` will collect average statistics per minute for a specific job (with job number="000000", job name="SCPF" and user name="QSYS"). At the same time, `proc.cpu.util[,,total,,avg5,]` will collect sum of CPU usage for all running jobs per 5 minutes.

Note 1. The parameters **name**, **user** and **subsystem** are case-insensitive.

Note 2. The returned value could be more than 100% on the multi-CPU systems. For example, if 2 jobs in the same set during the last period use processors 100% each, then result will be 200%.

Note 3. CPU utilization is calculated by the separate thread ("collector") that processes a job list every 3 seconds. So, short-lived jobs could be unprocessed by this thread.

proc.cpu.util.discovery[seconds]

Returns a JSON-object that lists all jobs already used at least specified **seconds** of CPU time. It could be useful for alerting about specific job if this job becomes to consume a lot of CPU resources. Each line contains the following macros (acceptable for `proc.cpu.util[]` metric): `{#NUM}` (job number), `{#USER}` (user name) and `{#NAME}` (job name).

proc.num[name,user,state,subsystem]

Integer – number of jobs with the given **name**, **user** and **state** in the given **subsystem**. All parameters are optional (default: all). The **subsystem** is regular expression (case-insensitive). **state** is a case-insensitive job state, it could be: "*ACTIVE", "*JOBQ", "*OUTQ", "RUN" (does mean "*ACTIVE" or "*JOBQ"), or any standard 4-character abbreviation of active job status (see description in API) like "sigw", "LCKW" or "hld " (note the trailing space!).

system.cpu.num[type]

Integer: the number of processors that are currently active in this partition. **type** could be "online", "max" or empty (in any case the same value is returned).

system.localtime[type]

Local system time according to Zabbix standard documentation. **type** should be "utc" (default value) or "local".

system.hostname

String with the AS/400 host name converted to lowercase.

system.uname

String with the AS/400 host name and version as well as JVM used.

system.users.num

Integer: the number of users currently signed on the system.

vfs.fs.discovery

JSON-string with the list of ASP's. Each discovered ASP has 2 macros: `{#FSNAME}` (ASP number, for example "1" for System ASP) and `{#FSTYPE}` (2-character according to the table below).

For example (for the System ASP):

```
{"data":[
 {"{#FSNAME}":"1","{#FSTYPE}":"00"}
]}
```

{#FSTYPE}	**Meaning**
00	system ASP
10	user ASP that does not contain libraries
11	user ASP that does contain libraries

vfs.fs.size[fs,mode]

Size of ASP specified by **fs** parameter (in bytes or percentage). All standard values for the **mode** parameter are supported: *total* (default), *free*, *used*, *pfree* (free, percentage) and *pused* (used, percentage).

vfs.fs.state[fs]

Integer that reflects a state of specified ASP according to the following table:

Value	**Description**
0	There is no status. This value is used for the system ASP and any basic user ASPs.
1	The status of the ASP is varyoff.
2	The status of the ASP is varyon.
3	The status of the ASP is active.
4	The status of the ASP is available.

as400.cpu.capacity

Float value: amount (in number of physical processors) of current processing capacity of the partition.

as400.disk.discovery

JSON-string with the list of physical disks or LUN's (for SAN-attached storage). Each disk is described by the following macros:

Macro's name	Description	Example
`{#DSK_SN}`	The serial number of the disk unit.	E0-D000038
`{#DSK_ID}`	Disk unit number. A unique identifier for each non-mirrored unit or mirrored pair among the configured disk units. Both mirrored units of a mirrored pair have the same disk unit number. The value of the disk unit number is assigned by the system when the disk unit is assigned to the ASP.	1
`{#DSK_TYPE}`	The type of disk unit.	2145
`{#DSK_MODEL}`	The model of the disk unit.	0050
`{#DSK_NAME}`	The unique system-assigned name of the disk unit.	DMP001
`{#DSK_ASP}`	ASP number the disk unit is assigned to.	1

as400.disk.asp[disk]

ASP number the disk unit is assigned to. The **disk** parameter must be disk unit's serial number (returned by the `{#DSK_SN}` macro during discovering).

as400.disk.size[disk,mode]

Size of disk unit specified by **disk** parameter (in bytes or percentage). The **disk** parameter must be disk unit's serial number (returned by the `{#DSK_SN}` macro during discovering). All standard values for the **mode** parameter of **vfs.fs.size[]** metric are supported: *total* (default), *free*, *used*, *pfree* (free, percentage) and *pused* (used, percentage).

as400.disk.state[disk]

Integer that reflects a state of specified disk unit according to the following table:

Value	Description
0	There is no unit control value.
1	The disk unit is active.
2	The disk unit has failed.
3	Some other disk unit in the disk subsystem has failed.
4	There is a hardware failure within the disk subsystem that affects performance, but does not affect the function of the disk unit.
5	There is a hardware failure within the disk subsystem that does not affect the function or performance of the disk unit.

6	The disk unit's parity protection is being rebuilt.
7	The disk unit is not ready.
8	The disk unit is write protected.
9	The disk unit is busy.
10	The disk unit is not operational.
11	The disk unit has returned a status that is not recognizable by the system.
12	The disk unit cannot be accessed.
13	The disk unit is read/write protected.
4294967295	Not configured or disk unit is not attached to this host

as400.outputqueue.size[outputQueue,library]

Number of spool files in the specified **outputQueue** in specified **library** (both parameters are mandatory).

as400.services[service]

Integer that reflects a state of one or several specified services (default – all). Returns zero if all checked services are available.

The service could be specified either by a keyword (see table below) or by an integer value (range 1 – 255). The integer is accepted as a bitmask where each bit means one of services (1 if it is needed to check).

Bit number	**Value**	**Service name**
0	1	FILE
1	2	PRINT
2	4	COMMAND
3	8	DATAQUEUE
4	16	DATABASE
5	32	RECORDACCESS
6	64	CENTRAL
7	128	SIGNON

So, you may specify both: either `as400.services[16]` or `as400.services[DATABASE]` to check the `DATABASE` service. At the same time, you can specify `as400.services[6]` to check the `PRINT` and `COMMAND` services. Default is 255 (check all services).

Returns the integer (0 – 255) where each bit corresponds to one of unavailable services. Accordingly, if all services are available, then zero will be returned.

as400.subsystem[subsystem,library]

String with the state of specified subsystem (usually "`*ACTIVE`" or "`*INACTIVE`"). Both parameters are mandatory.

as400.systemPool.discovery

JSON-string with the list of system storage pools. Each pool is described by the following macros:

Macro's name	Description	Example
`{#NAME}`	The name of system storage pool.	*BASE
`{#ID}`	The system pool numeric identifier (zero for inactive system pools).	2
`{#DESCR}`	The description of system pool.	The shared pool used for interactive work

NB! Inactive system pools could be absent in the list.

as400.systemPool.state[pool,mode]

Some property of the system pool specified by its name. The mandatory **pool** parameter should be a system pool name (as returned by the `{#NAME}` macro during discovering). The **mode** parameter could be one of the following:

- **size** (default if absent): integer value reflecting the amount of main storage (in bytes) currently allocated to the pool.
- **identifier**: unique integer identifier (1 – 64) assigned by system to each system storage pool that currently has main storage allocated. If the pool is inactive, 0 is returned.
- **description**: string containing the configured description of this storage pool.
- **databaseFaults**: float value that returns the rate, shown in page faults per second, of database page faults against pages containing either database data or access paths.
- **nonDatabaseFaults**: float value that returns the rate, shown in page faults per second, of nondatabase page faults against pages other than those designated as database pages.
- **totalFaults**: float value, the sum of **databaseFaults** and **nonDatabaseFaults**.

Implementation notes

My initial idea was: to write a Java-program emulating the behaviour of Zabbix-agent (as similar as possible). Zabbix product is open-source, so we always can investigate how some part was implemented there.

In theory, such program could run either on the AS400 system directly or on some other host and connect to AS400 via network (both modes have their pros and contras). So, from the Zabbix-server's perspective, such program looks like a usual Zabbix-agent. At the same time, this program can use Java

API and AS400 API to obtain needed information. Ideally if this program could support the most of standard Zabbix-agent's item keys (as far as they have sense for this architecture).

In reality, it uses both: IBM Toolbox for Java API (for the most metrics) and native AS400 QYASPOL API (for metrics relating to ASP's and disks). Native API returns a lot of information (full list of ASP's or disks along with all their details), so it is too ineffective to use it for each request and extract only small part of this info. Therefore, a caching is used: any request to ASP or disk causes to full result is cached for a 5 seconds, and all requests during that time are processes from the cache (without real API call). Expired cache is cleared. As a standard practice is to use the same or divisible intervals for related metrics, it dramatically reduces the load (especially for active mode). For example, if you have 6 disks and collect 6 metrics for each disk (plus LLD), then all these 37 metrics will be collected after a single API call.

Global regular expressions in item keys parameters are supported for the **eventlog** metric only. It could work in the **proc.num** metric also under the following conditions: 1) agent works in active mode; 2) the same global regular expression is used in some other **eventlog** metric. Unfortunately, the problem is not in the agent implementation, but in Zabbix protocol design (see https://support.zabbix.com/browse/ZBX-8295 for details).

Author

Constantin Oshmyan, COshmyan@gmail.com, available at Zabbix forum as Kos.

Credits

Hearty thanks to my colleague Ernests Daukšta for valuable ideas and consultations regarding AS/400 architecture.

Of course, this project could be impossible without a Zabbix team (open source and consultations with their specialists).

History

v0.4.0 (21.11.2016) First published version.

v0.4.1 (22.11.2016) Metrics **system.localtime**, **system.users.num** and **system.cpu.num** added.

v0.5.0 (06.12.2016) Metrics **proc.cpu.util** and **proc.cpu.util.discovery** added. Bug fixed: long reply from server (10KB and more) was not accepted correctly.

v0.5.1 (08.12.2016) Bug fixed: macro in active checks are processed correctly now. Enhancement: unexpected runtime errors in collector thread causes to immediate agent stop now (to preserve logs and show problem ASAP).

V0.6.0 (09.01.2017) Enhancements:

- Version of IBM Toolbox for Java API is written to log file during startup;
- Connections to AS/400 are permanent now (contrary to shirt-living connections before). It dramatically decreases the "noise level" of the agent (number of background jobs, their log- and spool-files, etc.).

- Auto-reconnect to AS/400 is implemented. Warning messages are written to log file only upon connection to AS/400 state changes: when it starts to fail and returns to normal (for each thread independently, however).
- If some active check fails, this failed result is returned to Zabbix server after first reiteration only; the first fail is ignored now. The old behaviour caused some metrics (like CPU utilization by jobs) switched to unsupported state immediately after the agent restart.
- Some small optimizations and log format cleaning.

V0.6.1 (25.01.2017) Bug fixed: the list handle returned by **QYASPOL** API is closed correctly now (using the **QGYCLST** API call). Bug symptom: the "Tried to go larger than storage limit for object" message in AS/400 job log.

V0.6.2 (22.02.2017) Output during initialization improved: errors after log-file initialization are written to the log file (for example in the case where jar-file for JSON parser is absent). Additional error messages added for debug.

V0.6.3 (07.03.2017) Workaround for non-appearing negative values of Job CPU Usage (API bug?).

V0.7.0 (25.05.2017) Enhancements:

- the auto-reconnection mechanism was improved;
- metrics **vfs.fs.state** and **as400.disk.asp** added;
- metric **as400.disk.state** improved (returns a fictitious value “4294967295” if the disk unit is not owned by the system);
- initialization and output were improved. The only place for any messages is a log file, if some problems with its accessing/creating – stdout. Presence of required libraries is checked explicitly. Values of metrics **agent.hostname** and **system.uname** are written to the log during initialization. In the case of any problems during initialization the program is stopped immediately.

V0.7.1 (29.06.2017) Enhancements:

- auto-reconnection mechanism improved: un-wrapping of any catched exceptions to check if it is an instance of **IOException** for checking on communication errors;
- active check thread modification for processing state when communication to AS/400 failed:
  - logging was cleaned (don’t spam a lot of messages);
  - do not send a failed check results to Zabbix-server (caused to switching to “not supported” state for these Item’s).
- initialization a bit improved (some additional checks).

V0.7.2 (06.07.2017) Enhancement: filtering of messages in message queue added (by a current user).

V0.7.3 (13.07.2017) Bug fixed: incorrect sequence of **wait()**, **notify()** and synchronized section could cause to dead-lock state in listener thread upon heavy-load condition. Enhancement: explicit call of

JobList.load() (as implicit call of this method from **jl.getLength()** does not throw an **IOExcepion** and returns invalid results instead).

V0.7.3.2 (13.11.2017) New metrics added:

- as400.cpu.capacity
- as400.systemPool.discovery
- as400.systemPool.state

V0.7.4 (06.03.2018) Bug fixed: **system.localtime** metric returned incorrect value.

V0.7.5 (26.04.2018) Bug fixed: correct version of really used **jt400.jar** library is logged during startup (previous versions erroneously wrote the version used during compilation, not at runtime).

V0.7.6 (19.10.2018) Bug fixed: correct processing of Zabbix protocol in passive mode (revealed with Zabbix Server v4.x).

V0.7.7 (04.10.2019) Bug fixes and optimizations, mostly in active check thread:

- bug fix (could cause to untimely agent shutdown if communication errors to Zabbix Server);
- bug fix (unneeded communications to Zabbix Server for **eventlog[...]** metric processing);
- optimization in **eventlog[...]** metric processing: if the last processed message could not be found in a message queue (for example, due to selective manual removal of some messages), then the Agent doesn't re-send the entire message queue again;
- optimization in stopping active check threads;
- addition of “id” for each value in sending JSON according to Zabbix 4.0 protocol;
- optimization in "proc.num[,,,subsystem]" metric;
- catching cases when some job disappears during **JobList** processing – just ignore them (**proc.num[…]** and **proc.cpu.util[…]** metrics).