REST Functions Supported by URS 8.0

This white paper is continuation of “Router Behind” API originally presented in 8.0.0 URS. This API is essentially a REST style one and not any SOAP counterparts are provided. The exposed functionality is divided into specialized sub-modules: interactions, statistics, diagnostic, entities. Normally any method is accessible through both HTTP GET and POST commands.

1. Interactions sub-module

All methods of interaction sub-module have prefix urs/call. Methods of this sub-module are used to start new (web) “interactions” and/or communicate with existing ones (no matter was they started as web one or are regular tserver based interactions). All methods from this section (except first one) require identification of interaction the action applied to. As such identification connection id is used and in method description it is marked as **<connid>.** As identification calls’ index can be used. Index value assigned to call from strategy and after that call can be identified by construction **[table:key]**(it supposed to used instead of connid in requests).
.

1.1 Start interaction/web session

This method instantiates new logical interaction(or session). As opposing to original RunStrategy method it is not wait until session will be over but reply immediately with connectionid of new starting interaction. Such interaction exists on its own and not internally bound to client that starts it, Client or any other component can communicate with this interaction on assumption it knows this interaction’s connection id.
urs/call/start?strategy=strategyname&tenant=tenantname&param1=value1&param2=value2

Input:
- tenant: name of tenant the new interaction will belongs to.
- strategy: optional (see note) name of strategy (Script object) that will be started.
- udata.somekey: optional, the value will just go into call’s attached data
- param: optional, any set of input parameters that will be available to running session as extensions (as opposed to udata keys) (can be accessed by strategy function ExtensionData)
- mediatype: optional, the value will just go into call’s attached data
- ani: optional, the value will just go into call’s ANI attribute
- dnis: optional, the value will just go into call’s DNIS attribute
- ced: optional, the value will just go into call’s CollectedDigits attribute
- thislocation: optional, the value define switch name for this call. If present on resource allocation phase URS will (at least try) to invoke ISCC GetAccessResource method on behalf of this switch name. ASssumption here is that URS has connection to TServer the resource belongs to and TServer is able to execute TGetAccessNumber invoked by external clients (like URS).

Positive output:
Connection Id of new session. Can be used later to communicate with this session.

Error
HTTP 400 code with body **Wrong request** if URS failed to start new session.

Note 1
If purpose of running strategy is allocation of available resource then strategy writer need to provide strategy code to report to client allocated resource. It can send for example some HTTP

message containing information about selected target. URS also provide some built-in functionality that might be used to address this task (see Note 2).

Note 2.
The following functionality allows using for web interactions **just the same strategies** (no need to rewrite a bit) that are working for regular tservers calls.
When running web session router gives special meaning of some data stored in interaction's extensions. This special data can be set through parameters of **urs/call/start** method as all parameters are going onto extensions of interaction or set directly in strategy with function ExtensionUpdate[key, value]. This special extensions keys are: replyurl, replybody, replyack and replyenc.
When (if) strategy execution allocate some resource to the current interaction and come to the point where it normally (for tserver interactions) issues routing request it check for existence in interaction extensions the value for key replyurl. If found then URS automatically sends HTTP request by this URL. The other 3 special keys can be used to set parameters of this HTTP request.

- replyurl: identify URL the URS will use to communicate back information about selected target. URL can include as its parts information about selected target.
- replybody: optional. If missed URS will use HTTP GET method. If provided HTTP POST will be used with this parameter value as POST message body. Body can include inside information about selected target..
- replyenc: optional. Has sense only if replybody provided. Used in context of replybody. This value will be used as value of Content-Type HTTP header in reply message. By default application/x-www-form-urlencoded is used.
- replyack: optional. Values are false or true, by default true. If set to true URS will wait confirmation on issues HTTP request. If error will be reported back then URS will consider that resource allocation failed and behave exactly in the same manner as if EventError was reported on RequestRouteCall. If replyack is false then URS will not wait any confirmation and blindly proceed strategy execution (do postrouting),

replyurl and replybody can (at least supposed to) contain inside the information about selected target. That achieved by using of the following formatting strings inside replyyurl and replybody:
[udata] or **[udata.somekey]** – extended to value of attached data (entire attached data or value of some specific key).
[ext] or **[ext.somekey]** – the same about interaction extensions.
[target], [target.*] or **[target.somekey]** – extended to value selected target (complete specification of attached data or some property of it like selected agent's name). Target specification has following properties:

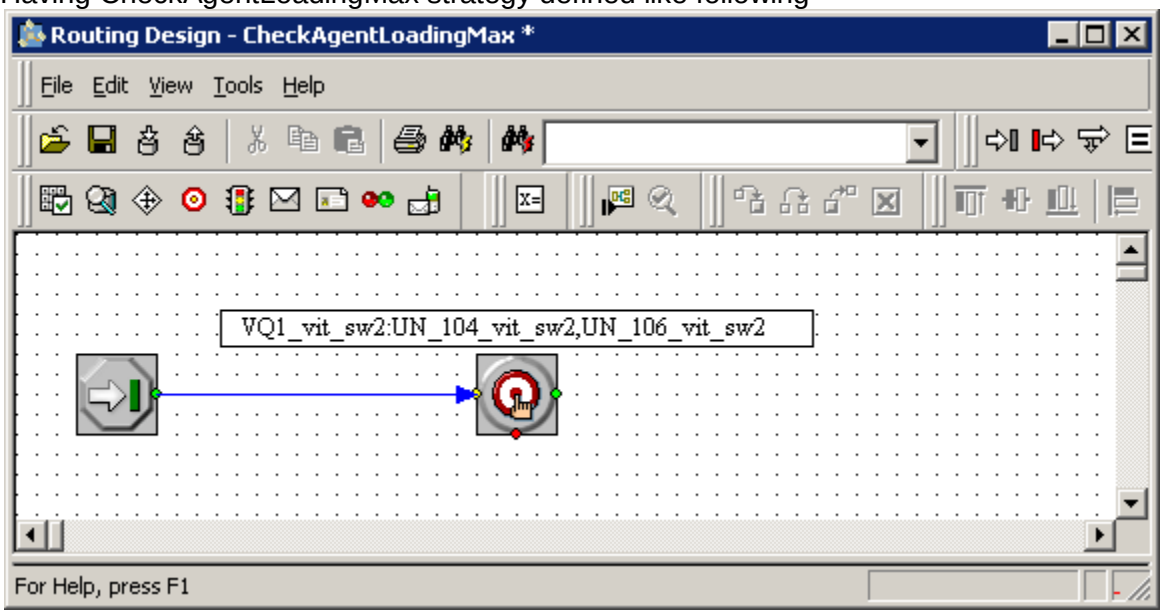
name – **return**, values – **default**, **direct**, **target**. Provide information how target was selected as default target, as result of force routing or normal target selection procedure.
name – **resource**, values – dn number call was routed to.
name – **dn**, values – dn number call was routed to.
name – **switch**, values – name of switch call was routed to.
name – **type**, values –type of target call was routed to: A, GA, etc.
name – **id**, values – name of target call was routed to, like name of agent group, etc
name – **agent**, values –agent the call was routed to.
name – **place**, values –place the call was routed to.
name – **vq**, values –name of virtual queue the selected target belongs to.
name – **stat_value**, values – statistic value the selected target has.
name – *, all listed above values in comma separated list of pairs in format name=value

no name at all, just like previous case but pairs are separated by & and always urlencoded
[targetj] - similar to **[target]** but presents all targets parameters in JSON format
[targetx] - similar to **[target]** but presents all targets parameters in XML format

Note 3: the parameter strategy is optional. If not specified the URS interpret request as request to find available target (what for else). In such case it runs predefined GetTarget logic. It expects that request parameters will contain all necessary for this strategy data specifically TargetList parameter:
http://localhost:2088/urs/call/start?tenant=Vit&**TargetList**=UN_104_vit_sw2.A,UN_106_vit_sw2.A&WaitTime=600&Statistic=StatTimeInReadyState&Criteria=1&VirtualQueue=VQ1_vit_sw2&Priority=10&replyurl=http://Barry7:8800/[target]

Sample.
http://localhost:2088/urs/call/start?tenant=Vit&strategy=CheckAgentLoadingMax&replyurl=http://Barry7:8800/[target]

Having CheckAgentLoadingMax strategy defined like following



as result of execution of this sample the following GET message might be sent Web service on Barry7:8800
http://Barry7:8800/return=target&type=A&id=UN_104_vit_sw2&agent=UN_104_vit_sw2&place=Place_104_vit_sw2&dn=104&switch=vit_sw2&resource=104&vq=VQ1_vit_sw2&stat_value=128

1.2 Sending request to interaction

This and following methods provide way to communicate with already running strategy
Assuming that interaction id is known (real or web interaction, doesn't matter) it is possible to send request to interaction and get back response. It is assumed that strategy for this

interaction contain code that expect external request and answer on them

urs/call/<connid>/request/name?param1=value1&param2=value2

Input:
- connid: id of interaction the request should be sent to.
- name: name of request. Name and parameters of request will be available to interaction.
- param: arbitrary set of keys with values. All parameters will be available to interaction

Positive output:
Whatever strategy will choose to return.

Error:
if call is not found then HTTP 404 code with body **Call not found.**
if strategy do not answer on request in promptly manner and stack of unanswered request becomes too big then HTTP 429 code with body **Overflow**
otherwise HTTP 400 code with body **Wrong request**.

Note 1.
Strategy also has option to return error code back instead of positive response.

Sample:
http://localhost:2088/urs/call/006b01fff974500b/request/tts?URL_ID=http://aaaaaaa

1.3 Sending event to interaction

Very similar to 1.2 Sending request to interaction but do not assume strategy should answer on it immediately. Instead strategy has option to process events later.

urs/call/<connid>/event/name?param1=value1&param2=value2

Input:
- connid: id of interaction the request should be sent to.
- name: name of request. Name and parameters of request will be available to interaction.
- param: arbitrary set of keys with values. All parameters will be available to interaction

Positive output:
OK

Error:
if call is not found then HTTP 404 code with body **Call not found.**
if strategy do not answer on request in promptly manner and stack of unanswered request becomes too big then HTTP 429 code with body **Overflow**
otherwise HTTP 400 code with body **Wrong request**.

Sample:
http://localhost:2088/urs/call/006b01fff974500b/event/tts?URL_ID=http://aaaaaaa

1.4 Terminating of strategy.

This action is sort of opposite to 1.1 Start interaction/web session. This method terminate running session and not the “physical” call itself. That means that this method more suitable for sessions started through web interface and not for real calls. Applying it to real calls is possible but should be performed with understanding – real call will continue to exist (potentially unprocessed)

urs/call/<connid>/terminate

Input:
- connid: id of interaction the request should be sent to.

Positive output:
OK

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/terminate

1.5 Strategy injection.

Methods **1.2 Sending request to interaction** and **1.3 Sending event to interactions** provide way to communicate with running for some call strategy. Their usage however means that strategy that process interaction should contain logic for this communication that make it more complex (if only not the case where only purpose of running strategy is answering web requests).

In some cases it is possible to communicate (run some logic) with running strategy transparently (invisibly) for main strategy that process the call. Specifically (but not limited to this cases only) it is situations when purpose to communicate with running strategy is just perform some corrective action (changing priority, targets, etc) or some queering of call status, etc.

The **invoke** action allows to interrupt running strategy, execute the provided subroutine for needed interaction (original strategy at this moment is frozen) report some data back and resume original strategy.
urs/call/<connid>/invoke?strategy=strategyname&param1=value1&param2=value2

Input:
- connid: id of interaction in context of which the subroutine will run.
- strategy: name of subroutine to run,
- param: optional, any set of parameters that might be passed to subroutine. Subroutine input parameters will be automatically initiated with parameters values taken from this request on

assumption parameters names match to name of subroutine input parameters.

Positive output:
JSON presentation of object having as properties the subroutine output parameters.

Error:
if call is not found then HTTP 404 code with body **Call not found.**
if URS fail to execute subroutine or subroutine returns error then HTTP 500 code with body **Internal error**

Note.
Set of possible functions that might be executed from inside subroutine is limited – URS raises error on attempt to execute any lengthy operation from inside such subroutine. Normally it means that is not possible to directly from subroutine body invoke operation like routing, applying treatments, querying external data source (database, web services, external services).

Sample:
http://localhost:2088/urs/call/006b01fff974500b/invoke?strategy=Add&arg1=10&arg2=20

1.6 Changing interaction attached data.

The method and few next one (1.7 Changing interactions extensions and 1.8 Accessing attached data) are sort of specialized form of affecting or querying the running strategy and might be considered as shortcuts for special cases achieved with strategy injection (see 1.5 Strategy injection).
The following method change interaction attached data (on fly). Might be thought of as remote execution of strategy function Update[key, value]

urs/call/<connid>/update?key1=value1&key2=value2

Input:
- connid: id of interaction the request should be sent to.
- keys: attached data to be set or changed.

Positive output:
OK

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/update?MyData1=123&MyData2=qwerty

1.7 Changing interaction extensions.

The following method change interaction extension data (on fly). Might be thought of as remote execution of strategy function ExtensionUpdate[key, value]

urs/call/<connid>/xupdate?key1=value1&key2=value2

Input:
- connid: id of interaction the request should be sent to.
- keys: extensions keys to be set or changed.

Positive output:
OK

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/xupdate?MyData1=123&MyData2=qwerty

1.8 Accessing attached data.

The following method query interaction attached data. Might be thought of as remote execution of strategy function UData[key]

urs/call/<connid>/udata?key1&key2&…

Input:
- connid: id of interaction the request should be sent to.
- keys: attached data to be set or changed.

Positive output:
JSON presentation of object having as properties the asked attached data keys..

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/xupdate?MyData1=123&MyData2=qwerty

1.9 Querying call.

The quick and having no side effects way to find out does URS process some call or not.

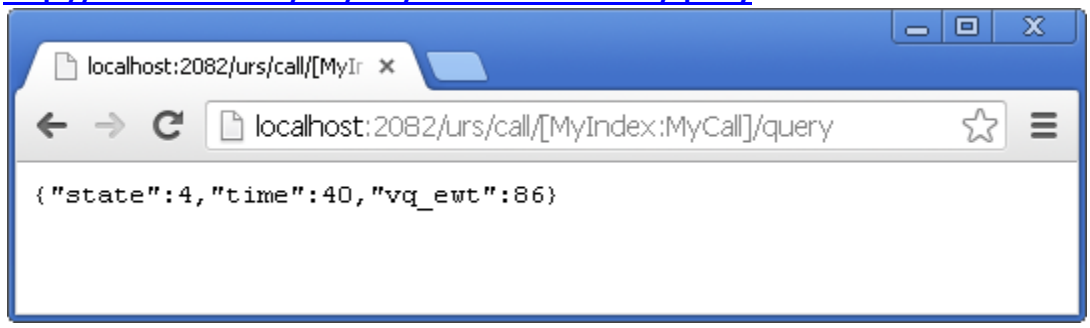
urs/call/<connid>/query

Input:
- connid: id of interaction the request should be sent to.

Positive output:
JSON presentation of object having 3 properties: current routing state of call and time in seconds URS run strategy for this call (absolute time, not CPU one).and expected waiting time for call to be routed (taken as minimal value among estimated times for all internal queues call is in)

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/query



1.10 Getting statistic data

While formally belonging to interaction sub-module this method effectively provide statistical information (in context of the call) and as such described in Statistics sub module (see 2.2 Call related sdata)

Sample:
http://routerhost:1234/urs/call/006b01fff974500b/sdata?stat=InVQWaitTime&targets=MyVQ.Q

1.11 Querying calls queuing information

This method is partially statistic related and partially is more detailed version of querying call (see 1.9 Querying call).

urs/call/<connid>/lvq?name=vqname&filter=string&ewt=min/max
outdated format: **urs/call/<connid>/lvq/name**?ffilter1&filter2 – not supported from 8.1.3

Input:
- connid: id of interaction the request should be sent to.
- name: (optional)name of Virtual Queue the request is related with. Also instead of name the simple name pattern (with wild char *) might be specified. If not provided the wildchar * will be used.

- filter: (optional) one or more Virtual queue properties to be reported like time, wt, etc. (see below properties of JSON object). If provided list of properties should be comma separated. If no filter is specified then all properties are reported.
- ewt: (optional). Can have values min or max. if provided only information about single virtual queue will be provided – the one having ewt property min or max.

Positive output:

JSON presentation of object having as properties the Virtual queue names the call is in (and that match to passed name parameter). Every such property has as value the object described position of call in this Virtual Queue. Such description has following properties:

time - UTF time stamp when call enters this Virtual queue,
wt - tome in seconds call is in this Virtual queue (effectively CurrentTime-time)
calls- current number of calls in this Virtual queue
pos – position of the current call in this Virtual queue
wpos – waiting position of the current call in this Virtual queue (different from pos in not counting calls which although still in queue but are already in routing state).
aqt – average quitting rate of calls from this Virtual Queue. If unknown this value will be skipped
ewt – expected waiting time for the call (effectively aqt * wpos). If unknown this value will be skipped
hit – percentage of call really distributed into this Virtual queue
guid – UUID of entering the call into this Virtual queue. If unknown this value will be skipped

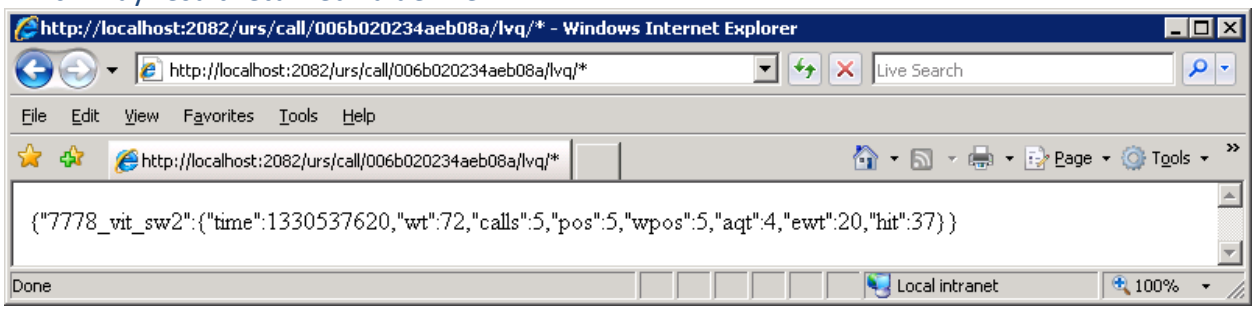
Error:

if call is not found then HTTP 404 code with body **Call not found.**

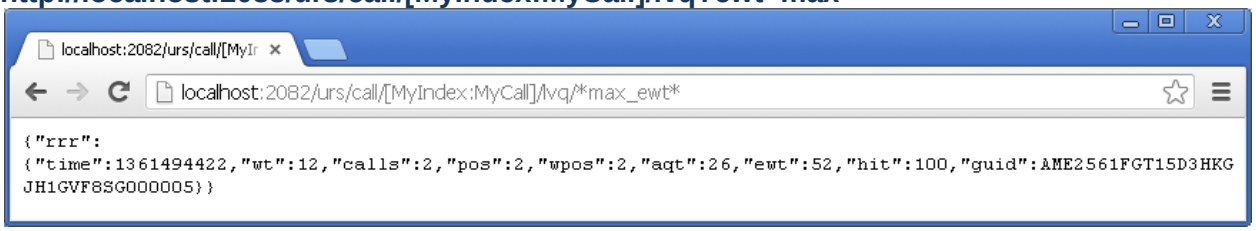
Sample:

http://routerhost:1234/urs/call/006b01fff974500b/lvq

Which may result returned value like



http://localhost:2088/urs/call/[MyIndex:MyCall]/lvq?ewt=max



1.12 Querying calls queuing information in individual internal queue

This method provides information about call position inside individual internal queue. Call can wait targets in multiple internal queues. Its entering inside one or another queue can be labeled (strategy function SetQueueLabel or through using included in [] prefix in virtual queue name).

urs/call/<connid>/rvqdata?id=label

Input:
- connid: id of interaction the request should be sent to.
- label: id of entering call into internal queue: either label or numeric id (if known).

Positive output:
JSON presentation of object having as properties different information about entering call into specific internal queue as well as some parameters of this internal queue.

aht - average handling time in this queue. It is calculated based on average handling time of every agent serving this internal queue.

pos – position of the call in this internal queue.

qlen – number of calls in this internal queue.

ewt – effectively value of aht multiplied by pos

qewt – effectively value of aht multiplied by qlen+1

quit – average quitting rate of calls from this internal queue. As opposing to aht it has sense for any internal queue (not only one having agents) but is based on short historical information

size – number of agents serving this internal queue

insize – number of logged in agents serving this internal queue

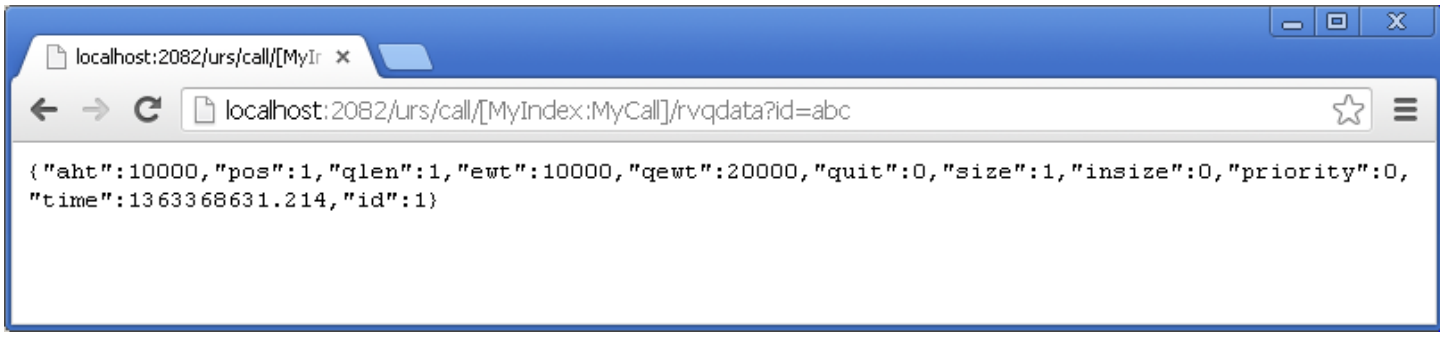
priority – priority of call in this internal queue

time - UTF time stamp when call was placed into this internal queue (with millisecond precision)

id – numerical id of this call placement in this internal queue

Error:
if call is not found then HTTP 404 code with body **Call not found.**

Sample:
http://routerhost:1234/urs/call/006b01fff974500b/rvqdata?id=abc
Which may result returned value like



1.13 Remote function invocation.

This method is simplified version of **1.5 Strategy injection** when instead of entire strategy just single function is invoked. This method is executed in the same way and the same restrictions are applied.

urs/call/<connid>/func?name=functionname&params=values

Input:
- connid: id of interaction in context of which the subroutine will run.
- name: name of function to execute,
- params: the JSON array of function parameters.

Positive output:
JSON presentation of value returned by function.

Error:
if call is not found then HTTP 404 code with body **Call not found.**
if URS fail to execute subroutine or subroutine returns error then HTTP 500 code with body **Internal error**

Sample:
http://localhost:2088/urs/call/006b01fff974500b/func?name=Dest&params=[]

2. Statistics sub-module

All methods of statistic sub-module have prefix urs/stat. Exception in sdata method formally belonging to interaction sub-module as accessing statistic data in context of some interaction.

2.1 Basic sdata.

This method is analog of strategy generic SData function.
urs/stat/sdata?statistic=statname&tenant=tenantname&targets=objectname

Input:
- statistic (or stat): name of statistic, exactly the same what strategy SData functions accepts.
- tenant: name of tenant targets and statistic definition belongs to.
- targets: coma separated list of targets. Every target is specified in the same format as used in regular strategies: name@statserver.type. Statserver and type parts are optional

Positive output:
JSON presentation of object where properties names match to name of targets and values are corresponding statistic value. Sample: {“Gropu1”:30, “Group2”:44}

Error:
HTTP 400 code with body **Wrong request**.

Sample:
http://routerhost:1234/urs/stat/sdata?stat=StatExpectedWaitingTime&tenant=Vit&targets=MyRP1.RP,MyRP2.RP

2.2 Call related sdata.

Formally the method belongs to Interaction sub-module, as requiring connid to operate.
Functionally however it is just the same sdata and closely related with “Basic sdata”.
This method is analog of generic strategy SData function as well as more specialized one: PositionInQueue, InVQWaitTime.
urs/call/<connid>/sdata?stat=InVQWaitTime&targets=MyVQ.Q

Input:
- connid: connection id of interaction in context of which sdata will be calculated.
- stat: name of statistic, exactly the same what strategy SData functions accepts.
- targets: comma separated list of targets. Every target is specified in the same format as used in regular strategies: name@statserver.type. Statserver and type parts are optional

Positive output:
JSON presentation of object where properties names match to name of targets and values are corresponding statistic value. Sample: {“Gropu1”:30, “Group2”:44}

Error:

if call is not found then HTTP 404 code with body **Call not found.**
otherwise HTTP 400 code with body **Wrong request**.

Sample:
http://routerhost:1234/urs/call/006b01fff974500b/sdata?stat=InVQWaitTime&targets=MyVQ.Q

2.3 Querying agent/place state.

Effectively this is still the same sdata but specifically asking TargetState/CurrentTargetState statistic which output is not scalar value but complex structure.
urs/stat/targetstate?tenant=tenantname&target=objectname&json=1

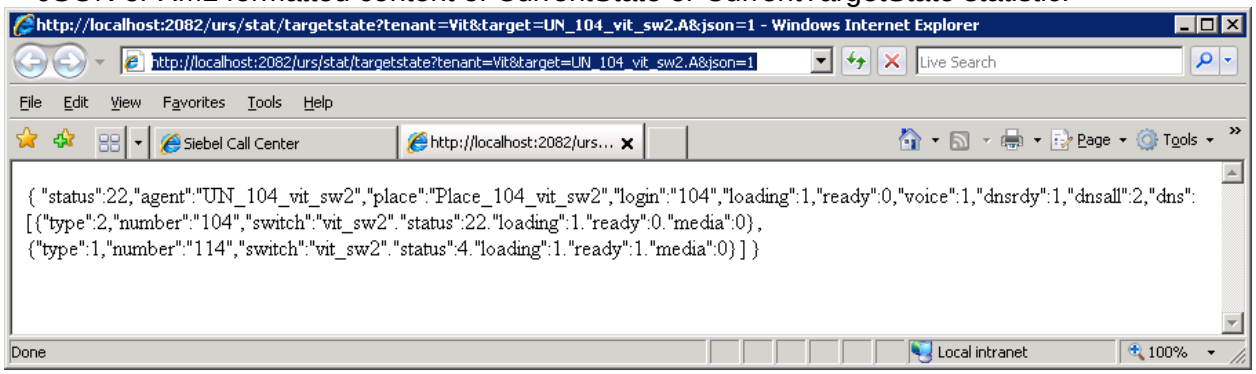
Effectively the method returns the same information as strategy function TargetState (see description in URS reference manual)

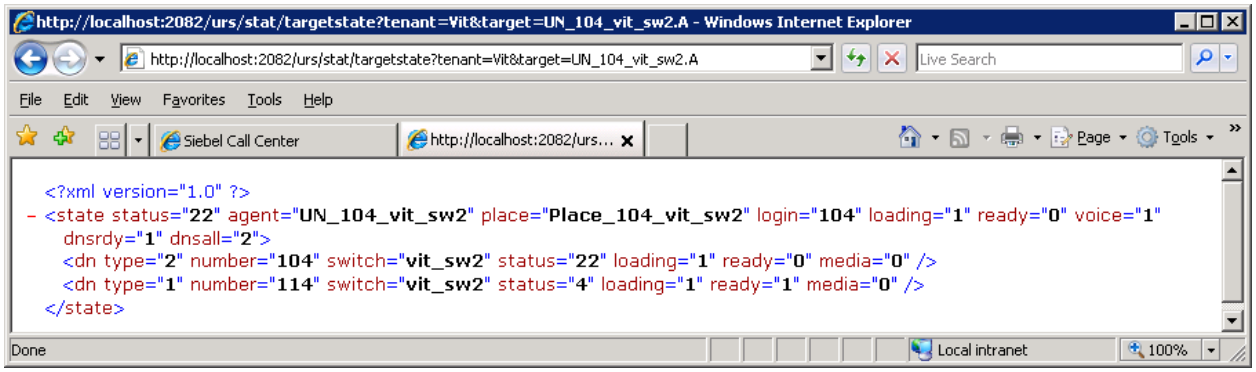
Input:

- tenant: name of tenant targets and statistic definition belongs to.
- target: coma separated list of targets. Every target is specified in the same format as used in regular strategies: name@statserver.type. Statserver and type parts are optional. Target should specify agent or place.
- json: optional, control output format. If present without value or having value 1 results output in JSON, otherwise output will be provided in xml format.

Positive output:

JSON or XML formatted content of CurrentState or CurrentTargetState statistic.





Error:

if target can not be identified then HTTP 404 code with body **Object not found.**
otherwise HTTP 400 code with body **Wrong request**.

Sample:
http://localhost:2088/urs/stat/targetstate?tenant=Vit&target=UN_104_vit_sw2.A&json=1

2.4 Querying router's queues.

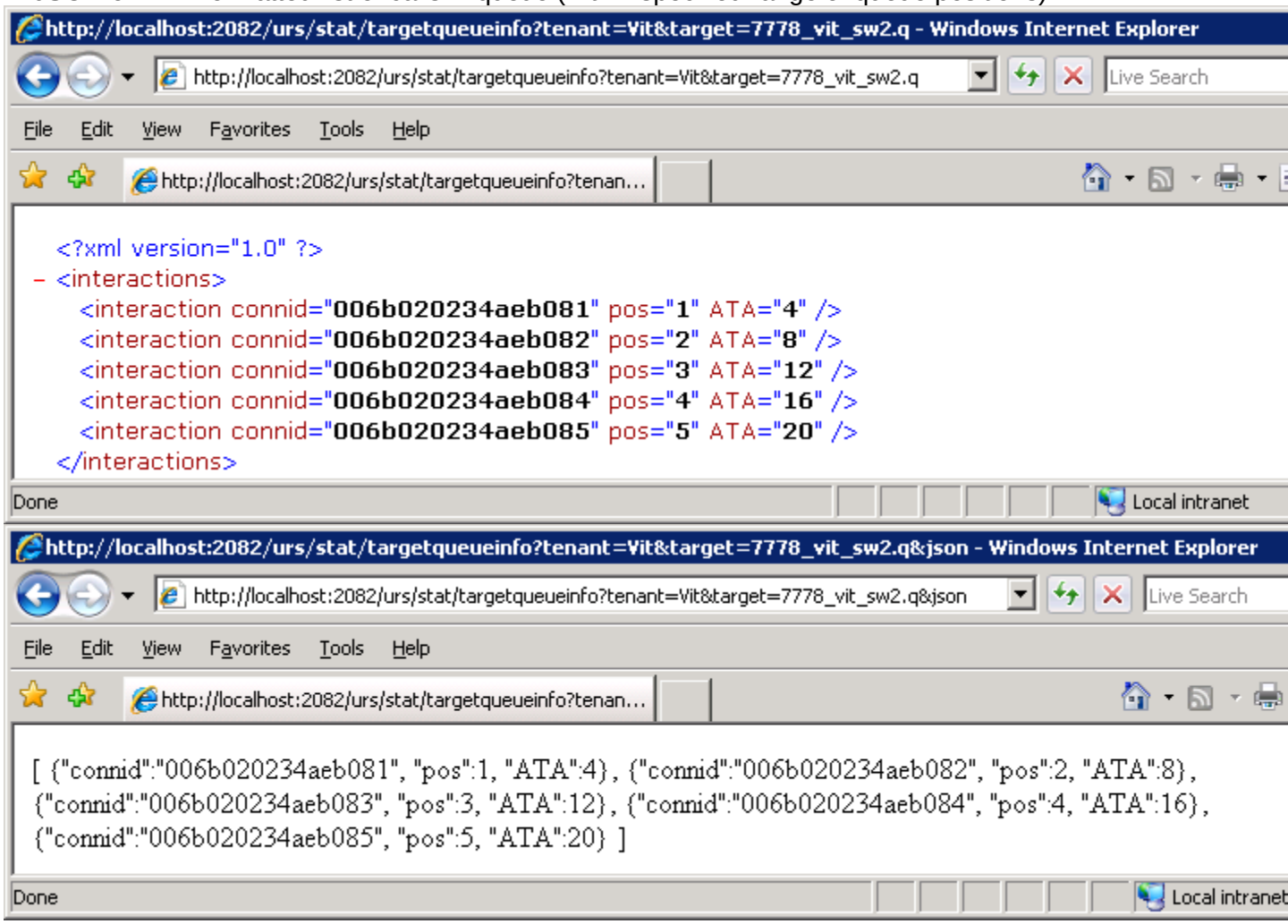
For every target URS keep queue of waiting it interactions. Content of the specific queue can be queried with targetqueueinfo method.
urs/stat/targetqueueinfo?tenant=Vit&target=MyAgentGroup.GA&range=<from>:<to>&json=1

Input:

- tenant: name of tenant targets and statistic definition belongs to.
- target: identifies target or virtual queue. Specified in the same format as used in regular strategies: name@statserver.type. For Virtual Queue target type is .Q.
- range: optional parameter. Default value 1:100. Specify range of queue positions. Only calls within this range will be reported.
- json: optional, control output format. If present without value or having value 1 results output in JSON, otherwise output will be provided in xml format.

Positive output:

JSON or XML formatted list of calls in queue (within specified range of queue positions)



Error:

if target cannot be identified then HTTP 404 code with body **Object not found.**
otherwise HTTP 400 code with body **Wrong request**.

Sample.
http://localhost:2088/urs/stat/targetqueueinfo?tenant=Vit&target=Agents1_5.GA

2.5 Statistic Subscription.

Alternative way to get any of statistic info described in 2.1 – 2.4 is subscription. As result URS itself notify subscriber about current statistic values. Subscriber should be able to act as web server to be able to receive URS notifications.

urs/stat/subscribe?statistic=statname&tenant=tenantname&target=objectname&connid=<connectionid>&interval=interval&replyurl=<notification url>&replybody=<body of notification message>&replyenc=<ecoding for reply body>&replyurl2=<alternative notification url>&cond=<conditon expression>&errorurl=<url to report errors>&expire=expiretime&range=range

Input:

- tenant: name of tenant targets and statistic definition belongs to.
- target: identifies single target the statistic will be applied to. Specified in the same format as used in regular strategies: name@statserver.type.
- statistic (or stat): name of statistic, exactly the same what strategy SData functions accepts. As exception the special names **TargetState**, **TargetStateJ, TargtetQueueInfo, TargetQueueInfoJ** (see 2.3 and 2.4) are accepted too.
- connid: optional, connection id of interaction in context of which statistic will be calculated. If not provided calculation will be done without any interaction context..
- replyurl: identify URL the URS will use to communicate back result. URL can include inside information about target, statistic value, etc (see Note below).
- replybody: optional. If missed URS will use HTTP GET method. If provided HTTP POST will be used with this parameter value as POST message body. Body can include inside information about target, statistic value, etc (see Note below).
- replyenc: optional. Has sense only if replybody provided. Used in context of replybody. This value will be used as value of Content-Type HTTP header in notification message. By default application/x-www-form-urlencoded is used.
- replyurl2: optional, identify reserved contact url. URS switch to this URL if it fail to report data to primary. URS reuse replybody and replyenc with this replyurl2.
- errorurl: if subscription was successfully started but has to be terminated due to external reasons this parameter provide URL that can be used to report such subscription termination. If not provided the subscription will be terminated silently. “External” reasons are termination of cal the subscription is associated with (see connid) or config object is removed from configuration.
- interval: optional, if not defined 0 will be used. Define time in seconds between subsequent notification messages. If set to 0 pseudo “change based” notifications approach will be used. URS will recalculate statistic every 2 seconds, compare value with previous one and if different the message will be sent.
- range: optional parameter. Default value 1:100. Has meaning only for subscription on target queue info (statistic is **TargetQueueInfo**(**J**), see 2.4 querying routing queues).
- expire: optional parameter. If provided (interpreted as UTF timestamp) define moment of time subscription should be terminated.
- cond: optional, if provided will define “threshold expression” that should be evaluated to true for URS to send notification message. The expression language is the same as used in IRD functions like SetTargetThreshold (see description in URS reference manual). This parameter doesn’t affect reevaluation of statistic itself but only sending of notification message to subscriber.

Positive output:

Subscription ID. Can be used in notifications messages.

Error

if target cannot be identified then HTTP 404 code with body **Object not found.**

if connid is provided but call can not be found then HTTP 404 code with body **Call not found**

if cond is provided but its parsing failed then HTTP 404 code with body **Wrong expression** otherwise HTTP 400 code with body **Wrong request**.

Note 1.
URLs and body of notifications messages are defined in subscription request itself through parameters like **replyurl**, etc. They supposed to be valid urls and expected POST contents containing formatting strings that will be replaced (I the process of building notification message) with their values:

[value] – replaced with current statistic value. Statistic value for most statistics is plain number. If subscribed statistic is TargetState the value is XML formatted output of target state (see 2.3 Querying agent/place state). If subscribed statistic is TargetStateJ the value is JSON formatted output of target state (see 2.3 Querying agent/place state). If subscribed statistic is TargetQueueInfo the value is XML formatted output of queue content (see 2.4 Querying routing queues). If subscribed statistic is TargetQueueInfoJ the value is JSON formatted output of queue content (see 2.3 Querying agent/place state).

[old] – replaced with old statistic value (calculated on previous step). It is no applicable to subscription on TargetState, TargetStateJ TargetQueueInfo and TargetQueueInfoJ.

[tenant] – replaced to tenant name from subscription request

[statistic] – replaced to statistic name from subscription request

[target] – replaced to target name from subscription request

[id] – replaced by value of subscription id (the one returned by HTTP subscription request)

[reason] – has sense for errorurl and is replaced with description of error. Possible values: Call not found, Object not found

[connid] – replaced to connid from subscription request

[n] – incremental counter of notification messages

[opt.optionname] – replaced with value of router's option optionname. For example can be used to detect that backup URS is now in charge of sending reporting messages.

Note 2.
Every notification message sent by URS should be confirmed (with HTTP 200 response).
If notification result error message received by URS then URS will try to switch next time to backup url (replyurl2) if provided. If URS fails to get at least one positive response (from either replyurl or replyurl2) for more then 3 minutes then this subscription is silently terminated.

Sample.
http://localhost:2088/urs/stat/subscribe?statistic=StatTimeInReadyState&tenant=Vit&target=UN_106_vit_sw2.A&interval=10&replyurl=http://Barry7:8800/value:[value]|id:[id]|opt:[opt.transition_time]

http://localhost:2088/urs/stat/subscribe?statistic=StatTimeInReadyState&tenant=Vit&target=UN_106_vit_sw2.A&replyurl=http://Barry7:8800/value:[value]|id:[id]|old:[old]

http://localhost:2088/urs/stat/subscribe?statistic=($(StatTimeInReadyState)*2.3)&tenant=Vit&target=UN_106_vit_sw2.A&replyurl=http://Barry7:8800/value:[value]|id:[id]|old:[old]

http://localhost:2088/urs/stat/subscribe?statistic=($(StatTimeInReadyState)*2.3)&tenant=Vit&target=UN_106_vit_sw2.A&replyurl=http://Barry7:8800/value:[value]|id:[id]|old:[old]&replybody=value:[value]]

http://localhost:2088/urs/stat/subscribe?statistic=StatTimeInReadyState&tenant=Vit&target=UN_106_vit_sw2.A&interval=10&replyurl=http://Barry7:8800/value:[value]|id:[id]&cond=sdata(Agents5_10.GA, StatAgentsAvailable)>1

http://localhost:2088/urs/stat/subscribe?statistic=TargetQueueInfo&tenant=Vit&target=Agents1_5.GA&interval=10&replyurl=http://Barry7:8800/value:[value]]

2.6 Statistic unsubscription.

Used to explicitly cancel subscription (see 2.5 Statistic Subscription). There are 2 variation to unsubscribe – through subscription id or through subscription definition.

urs/stat/unsubscribe?id=id
urs/stat/unsubscribe?statistic=statname&tenant=tenantname&target=objectname&interval=interval&replyurl=<notification url>&connid=<connectionid>&range=range

Input:
- id: id of subscription. This value was returned by original statistic subscription request
- tenant: tenant from subscription request
- target: target from subscription request
- statistic (or stat): statistic from subscription request
- connid: optional, connection id from subscription request
- replyurl: replyurl from subscription request.
- range: optional, range from subscription request

Note. Alternative way to unsubscribe from statistic is stop to confirm statistic notifications sent by URS.

Positive output:
OK

Error
if subscription cannot be found based on provided parameters then HTTP 404 code with body **Object not found.**

Sample.
http://localhost:2088/urs/stat/unsubscribe?id=1
http://localhost:2088/urs/stat/unsubscribe?statistic=StatTimeInReadyState&tenant=Vit&target=UN_106_vit_sw2.A&replyurl=http://Barry7:8800/value:[value]|id:[id]&interval=10

2.7 Routing performance report.

Effectively duplicates corresponding command from Diagnostic sub-module (see 3 Diagnostic sub-module). Replicated in Statistic Sub-module to obtain this information in structured way (XML) as opposed to plain unstructured text diagnostic sub-module provide.

The method provide information about how many interaction was processed by URS at all or on specific tserver or routing point and how, how many of them was abandoned, etc

urs/stat/report?tserver=name&cdn=digits&count
urs/stat/report?ar

Input:
- tserver: optional – name of tserver for which one information is required.
- cdn: optional – number of routing point for which one information is required.
- count: optional – if present make URS report short information only (no time based counters).
- ar: optional – provide information about agent reservation effectiveness.

Positive output:
XML formatted information about URS performance.

Error
NA

Sample.
http://localhost:2088/urs/stat/report?tserver=vit_ts2&cdn=2203

will result

```
<statinfo>
  <tserver name="vit_ts2" switch="vit_sw2">
    <cdn digits="2203">
      <now>0</now>
       <handled>
         <calls>4</calls>
         <time total="156" n="23" t="0" x="0" s="0" w="117" f="0" r="11" />
       </handled>
       <routed>
          <calls>4</calls>
          <time total="156" n="23" t="0" x="0" s="0" w="117" f="0" r="11" />
      </routed>
      <confirmed>
        <calls>4</calls>
        <time total="156" n="23" t="0" x="0" s="0" w="117" f="0" r="11" />
      </confirmed>
      <abandoned><calls>0</calls></abandoned>
      <error><calls>0</calls></error>
      <default><calls>0</calls></default>
```

```
        <dropped><calls>0</calls></dropped>
      </cdn>
    </tserver>
  </statinfo>
```

http://localhost:2088/urs/stat/report?tserver=vit_ts2&cdn=2203&count

will result

```
<statinfo>
  <now>0</now>
  <handled>4</handled>
  <routed>4</routed>
  <confirmed>4</confirmed>
  <abandoned>0</abandoned>
  <error>0</error>
  <default>0</default>
  <dropped>0</dropped>
</statinfo>
```

http://jcizas:2082/urs/stat/report?ar

will result

```
<agent_reservation_info>
  <exp_reqs>2</exp_reqs>
  <imp_reqs>0</imp_reqs>
  <pos>2</pos>
  <neg>0</neg>
  <used>2</used>
</agent_reservation_info>
```

Note. meaning of fields in this XML doc is (**completed** call here is one that URS already remove from his memory):

statinfo:

now – number of calls in routers memory for which URS started strategy execution.
handled – number of completed calls for which URS run some strategy.
routed – number of completed calls for router issued RequestRouteCall to selected target
confirmed – subset of routed (include only calls for which confirmation event (EventRouteUsed) was received).
abandoned – number completed calls that was abandoned
error – number of completed call calls for which at least one error was raised in process of running strategy
default – number of completed call which was sent URS to default destination
dropped – number of completed calls for which router has to terminate processing in the middle (but not because of abandon) - call was routed by switch, somebody remove RP from configuration or router disconnects from tserver etc.

agent_reservation_info

exp_reqs – number of explicit agent reservation requests (during last 1 min)
imp_reqs – number of implicit agent reservation requests (during last 1 min)
pos – number of positive responses for explicit agent reservation requests (during last 1 min)
neg – number of error for explicit/implicit agent reserving requests (during last 1 min)
used – number of processed explicit agent reservation requests (during last 1 min)

3. Diagnostic sub-module

Diagnostic sub-module essentially is HTTP enabled “console commands” interface with URS. Originally exposed as direct console command and used to browse and check states of internal object in URS (config object, stat objects, strategies, etc), setting not standard logging options, etc.

3.1 Router console access.

Invoke routers console command (the same one as possible to enter directly from URS console windows). To get list of console command the help console command (?) can be used

urs/console?command

Input:
- command: console command.

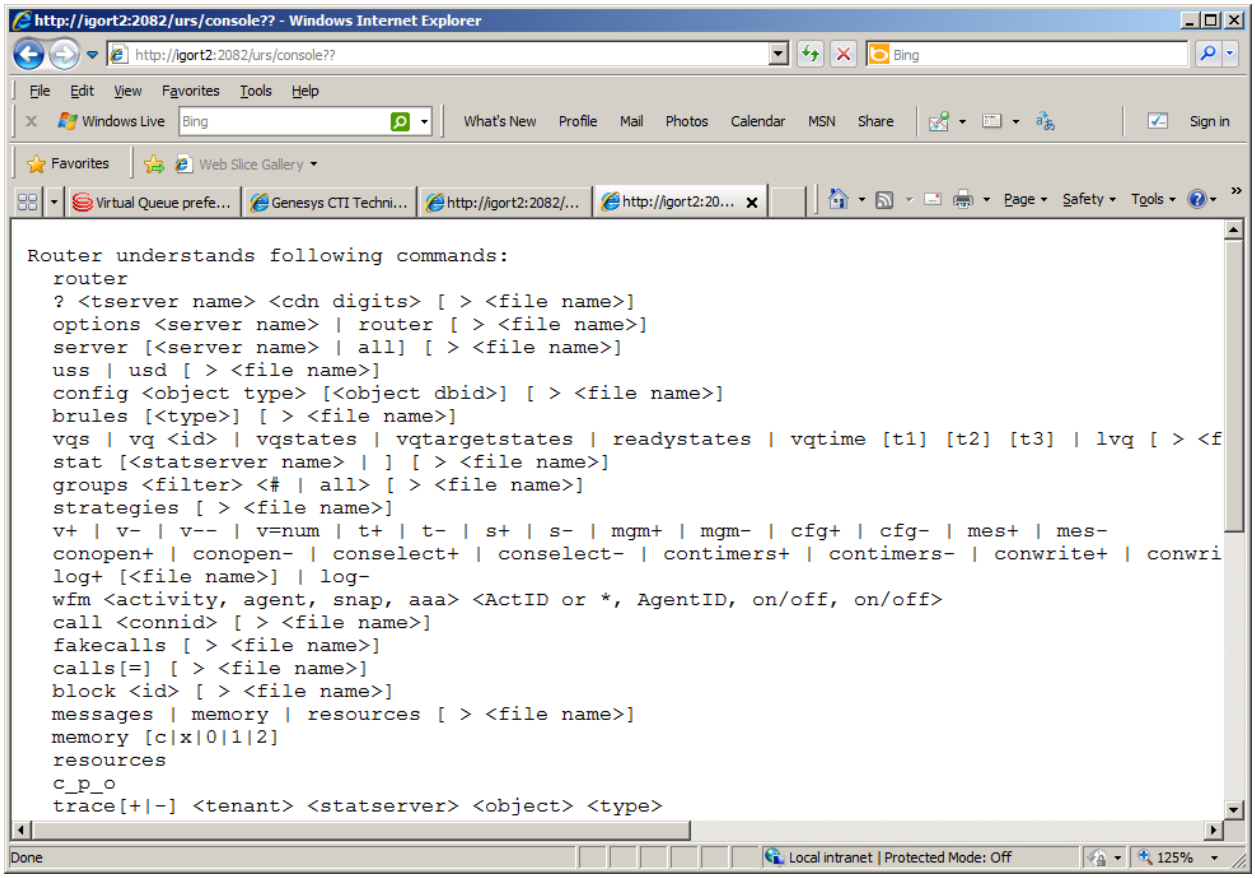
Positive output:
Depends from entered command.

Error
If URS do not understand it provide back HTTP 200 message with body **don’t understand**: <user input here>

Note. As some console commands can affect volume of log output or activate special mode of operation (like memory leaks detection) should be used very cautiously in production environments.

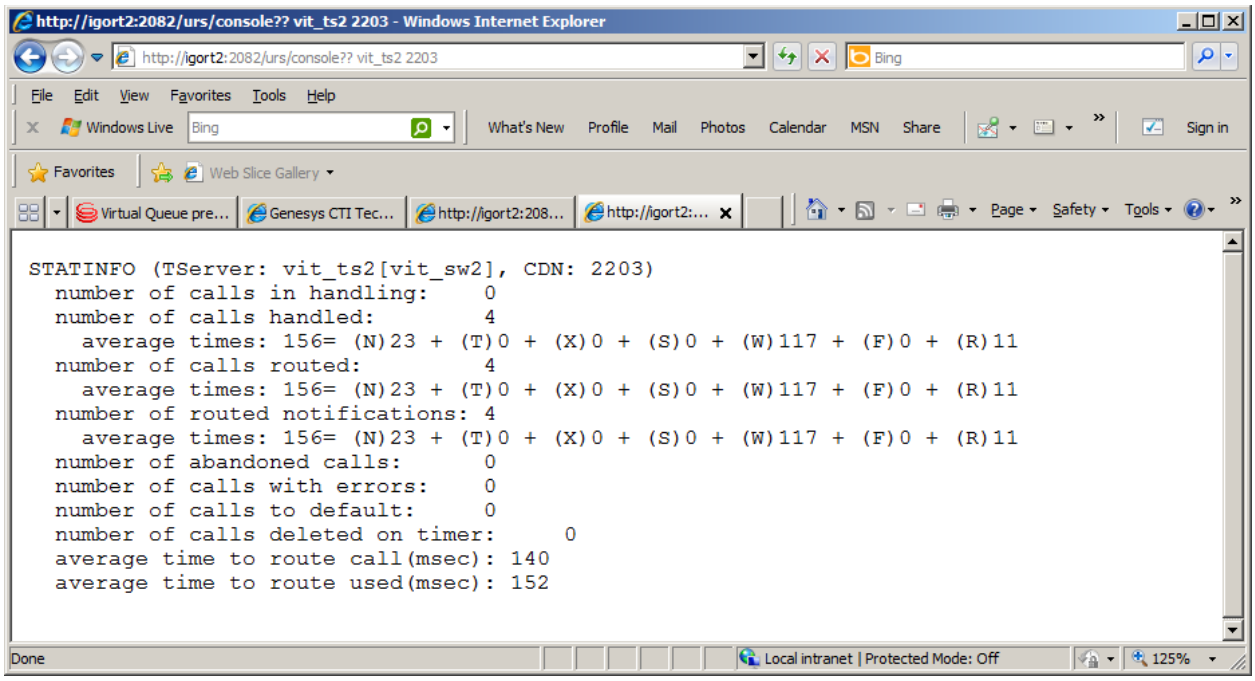
Sample.
querying list of console commands (command ?)
http://localhost:2088/urs/console??

Will result something like



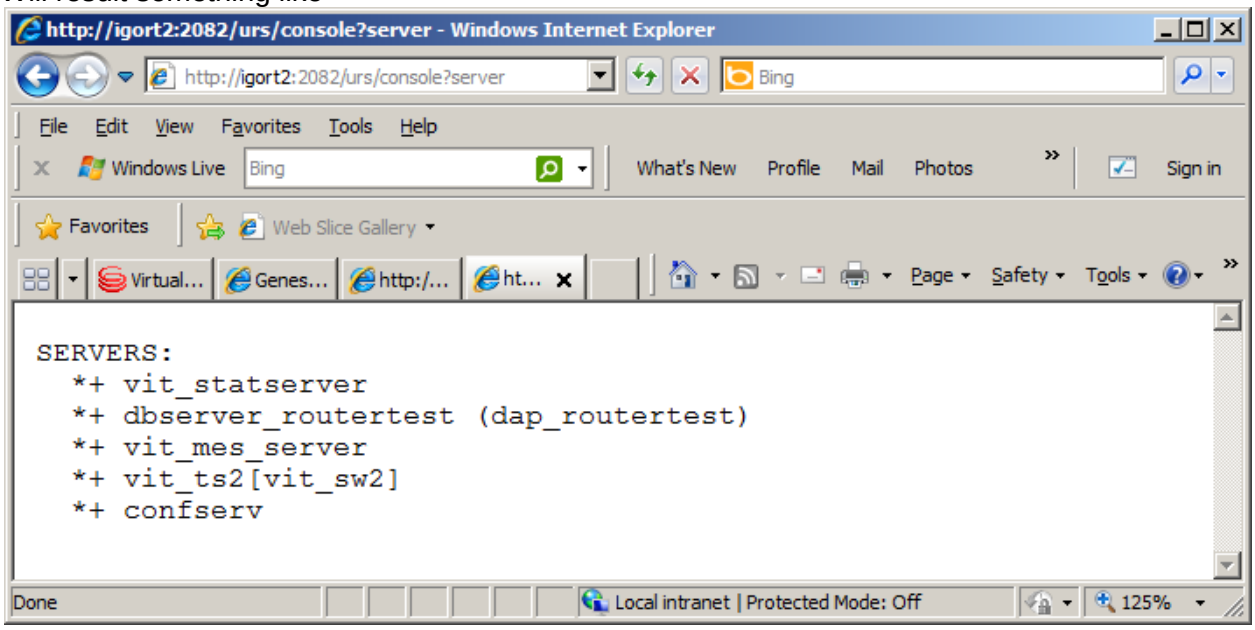
Sample.
Analog of 2.7 Routers performance request

http://localhost:2088/urs/console?? vit_ts2 2203
Will result something like



Sample.
Server's URS is connected to

http://localhost:2088/urs/console?server
Will result something like



4. Objects (entities) sub-module

Objects sub-module provides access to objects the router's strategy working with. That might me agents, places, other types of targets, switches, etc.

4.1 Targets tagging.

This method is used to mark/unmark agents or places with specific named tag. Such tags can be checked in strategy and routing decision can be taken based on presence or absence of some of them. They even can directly control "definition" of agent/place/switch readiness (see strategy function SetReadyCondition).

urs/entity/tag?tenant=tenantname&target=objectname&tag=name&expire=interval

Input:
- tenant: tenant the referred target belongs to
- target: target to be tagged or untagged
- tag (or name): optional, name of tag
- expire: optional, time in seconds the tag will be effective. If absent or set to 0 then tag has no expiration and if needed have to be untagged explicitly. If value is negative the target will be untagged. Any positive value will result the target tag will be self-cancelled in provided number of seconds.

Positive output:
JSON array of all tags the target currently have (after execution this command)

Error
If referred target cannot be found then HTTP 404 code with body **Object not found.**

Sample.
http://localhost:2088/urs/entity/tag?tenant=Vit&target=UN_106_vit_sw2.A
http://localhost:2088/urs/entity/tag?tenant=Vit&target=UN_106_vit_sw2.A&tag=rdnd
http://localhost:2088/urs/entity/tag?tenant=Vit&target=UN_106_vit_sw2.A&tag=bbbb&expire=60
http://localhost:2088/urs/entity/tag?tenant=Vit&target=UN_106_vit_sw2.A&tag=rdnd&expire=-1

Note. There is special tag with name **rdnd**. Its processing is hardcoded in URS and any target (agent or place or switch) having it are considered by URS as "not disturbable" – no call will be routed to them until this tag is in effect.

Note. The tags also can be set/cleared from strategy (the function SetTag[target, tag, expire]).

Note. The following function called in strategy SetReadyCondition['tag(bbbb)'] will result that current call might be routed only to agent having active tag **bbbb**. The same but opposite effect has invocation of SetReadyCondition['untag(bbbb)'].