# System Management, Phase I (Delhi Release), Design

Version 8 (and final pending last minute changes), 8/21/18

## Microservice SM Agent (SMA)

### Design Considerations

#### *SMA is an optional service*

We have to accept that the system management agent (and some of the API in the sys management API of each service) may be provided by more industrial / enterprise capability like Kubernetes or cloud deployment/orchestration facilities. So our platform allows the agent/APIs to be disabled.

EdgeX services, which provide the SM API, must have a configuration setting that disables the API. This ensures that rouge processes cannot use the SM APIs to bypass alternate system management frameworks.

#### *How to determine "all" services*

The system management agent (SMA) must know what services it manages and where each service is located. It often makes calls to all of the EdgeX microservices (ex: to stop all services). How is "all services" defined or determined? When Consul is running, it can provide a catalog of registered services, but this exposes a chicken and egg problem. If the SMA is requested to restart all services, for example, it could use Consul information to get all services and issue a stop command to each, but how would it be able to then send a start command to the mechanism required to start the service (as services cannot start themselves) if Consul is not even up yet?

Further, EdgeX has been created to operate without a registry/config service – especially for developer environments. Can the SM Agent operate without config/registry?

In order to facilitate the independent operation of SM and one that may bootstrap all of EdgeX, the SMA will be provided configuration (a manifest) that specifies the list of the services it manages. As with all services today, a bootstrap property (like –consul) will indicate to the SMA to get this configuration from the local file system or from Consul. The config will list the services that the SMA is managing, along with details on how to start/stop each service.

#### *SMA Manifest - Service Registry and Start/Stop Commands*

As the SMA must be able to start and stop each service (restart too but this is just a combination of stop and start), it must know how to issue the appropriate commands to start and stop each service. Unfortunately, the start and stop of each service may be directly associated to containerization (Docker or snap start/stop commands), service technology (Java, Go, etc.) and many other factors.

While EdgeX can attempt to standardize how its services are managed, there are associated 3rd party infrastructure services (MongoDB, Consul, Kong, etc.) that will not adhere to EdgeX guidelines. In fact, these services will have to be managed differently in that they will also not be able to respond to the SM API set.

A manifest configuration must be provided to the SMA (either via Consul or the local file system as indicated by the bootstrap property). This file must contain the names of the services, the intended location or address of the services (which for now would always be on the same host), and the commands to issue in order to start/stop the service. The manifest will also signify when the service is an EdgeX microservice (and conforms to the SM API) or if the service is not (MongoDB, Consul, etc.) and therefore not be subject to standard EdgeX API calls.

The manifest configuration will boostrap the SMA. Different versions of the file may exist depending on how/where EdgeX is deployed (Docker v. Snappy, Windows v. Linux, etc.).

In this first release of the SMA, the manifest will be static. In the future, the manifest may be more dynamic or even provided by some 3rd party orchestrator.

## System of record for configuration

While Consul is generally regarded as the home for configuration information, each service gets its initial configuration from either Consul or its local configuration file depending on startup parameters. So, in reality, only each service knows what its real configuration. Therefore, the SMA must request configuration information directly from the services - and allow the configuration to come from Consul or a local configuration file as deemed by that service.

### *Metrics Significant Change*

On any dynamic metric, like memory usage, clients that have interest in the change are usually interested in “significant” change. But the significance may be environmentally or client dependent. Therefore, on startup of any service, a configuration property must be specified that indicates the min/max range for the metric. The range can be different per service and per metric. Registered clients are notified by the service when the metric is outside the range.

### *Metrics of Interest and Implementation*

Over time or depending on use case/configuration, the metrics of interest will change. In the Delhi release, the community plans to support only one or two metrics to prove out the APIs and means of obtaining the metric. The SMA (and services) should allow for configuration as to which metrics are available. The API must be generic enough to accept a request for any metric by parameter (ex: getMetric{“memoryUsage”} ) versus having a predefined method (ex: getMemoryUsage{} ) so as to allow the accessible metrics to change as the service capability evolves or as use cases dictates.

The metrics themselves may be determined in many different manners. In some cases, the service itself may provide the metric. In other cases, the container or underlying OS is the mechanism for determining the metric. Therefore, the SMA should contain a layer of abstraction that allows for different implementations. A reference implementation may assume a certain means of getting the metric(s), but this may change in certain environments or microservice implementations. Further, under the covers of the abstraction, the frequency, caching of metric results, etc. may be configurable and fine-tuned for use case, environment, etc.

## SM Agent API

Note: All REST API call body and responses are in JSON.

The SMA must be able to respond to the following public REST APIs

- Stop/start/restart EdgeX microservice(s)
    - api/v1/operation {via POST}
    - The body of the request must specify the operation action (start, stop, restart), optional parameters (example: force or graceful) and name of the microservice or microservices

    targeted for the operation.  Example:  { “action”: “stop”, “params”: [“force”:true], “services”: [“edgex-core-data”, “edgex-core-metadata”, …] }
  - The parameter list is optional and open ended.
  - The API returns 200 (ok) status on successful internal call to actions on all services, or an array of failed actions on particular services (see more below).

- Get all of the configuration settings (aka properties) for EdgeX microservice(s) by name
  - api/v1/config {via GET}
  - The body of the request must specify the name of the microservice or microsesrvices of interest.  Example: { “services”: [“edgex-core-data”, “edgex-core-metadata”, …] }
  - The return result will be an array of service:config results.  Example: [{“service”:“edgex-core-data”, “config”:[“port”:48080, “loggingLevel”:”debug”…]}, {“service”:“edgex-core-metdata”, “config”:[“port”:48081, “loggingLevel”:”error”…]}]
  - Note: the structure of the configuration data will be returned in JSON as it is provided by the micro service.  However, the structure will likely be more hierarchical (versus simple key/value pairs shown above) given the changes being made to configuration for Delhi.
- Note:  the project consider the Hateos pattern:  https://spring.io/understanding/HATEOAS but felt this feature was really meant for more semantic web style of services (OpenAPI and binding generation).  May be helpful to things like UIs where further details need to be accessed and links provide the means to get it.  Not including, but subject to re-opening with use case demand of if additional rationale can be provided.

- Get the configuration setting (single property versus all properties per API above) for EdgeX microservice(s) by name
  - api/v1/config/[config property name] {via GET}
    - example:  api/v1/config/port
  - The body of the request must specify the name of the microservice or microsesrvices of interest.  Example:  { “services”: [“edgex-core-data”, “edgex-core-metadata”, …] }
  - The return result will be an array of service:config results.  Example: [{“service”:“edgex-core-data”, “config”:[“port”:48080]}, {“service”:“edgex-core-metdata”, “config”:[“port”:48081]}]
  - Note: the structure of the configuration data will be returned in JSON as it is provided by the micro service.  However, the structure will likely be more hierarchical (versus simple key/value pairs shown above) given the changes being made to configuration for Delhi.

- Setting the configuration (aka property) for a writable (versus read only property for an EdgeX microservice) is outside the scope of this first release.

    - How would we enforce read v. write settings?  Should they all be read/write?  Something to be covered for the next release.

- Get metrics an EdgeX microservice(s) by name
    - api/v1/metric (via GET)
    - The body of the request must specify the metrics of interest (memory, CPU, …) and name of the microservice of concern.  Example:  {“metrics”:[“memory”, “CPU”], “services”: [“edgex-core-data”, “edgex-core-metadata”, …] }
    - The return result will be an array of metric results per service.  Example: [{“service”:“edgex-core-data”, “metrics”:[“memory”:”34MB”, “CPU”:”3%”]}, {“service”:“edgex-core-metdata”, “metrics”:[“memory”:”31MB”, “CPU”:”2%”]}, …]

    Note:  the project discussed the possibility of adding a unit of measure to the response.  It was determined that the client should know what is returned and that the documentation should provide unit of measure information and understanding.  In other words, follow a keep the JSON light; no-schema approach.

    Because of challenges on the DS SDK and precision issues, there was a consideration as to whether we need to worry about precision in the values returned. Do we, for example, need to worry about representing some native numeric representation in strings form?  It was determined that the metrics were really a “gross” representation (since some metrics are going to be OS, environment, or containerization dependent) and not precise.  We may need to readdress this issue if use cases start to require more precision and change our requirements.

- Getting/setting admin or operational status of microservices is beyond the scope of this first release.  It only applies to device services at this time.
- Callbacks (and associated registration/deregistration for changes to status, metrics, configuration, etc. is out of scope for this first release of EdgeX.
    - As this is a highly desired feature, we may relook and implement part of this for Delhi if time perhaps.

*Note: a Ping or current status API was considered but rejected because it duplicated functionality provided by Consul/service APIs and it may be handled by Docker, Snappy, etc. facilities.*

## SMA API Implementation

Implementation assumed in Go and therefore expressed in Go idioms.  Gorilla Mux or other router, directs the REST client requests to these methods.  An SMA package (sys-management-agent)defines these functions:

ServiceOperation(action string, parameters []string, services []string)

For each service in the services list, use the local manifest (likely cached on bootrstrap) to issue the appropriate start, stop or restart command as dictated by the action, and passing in the parameters as appropriate.

**Implementation**

- For ea: service
  - Make asynchronous call (go routine) to appropriate internal function (Stop, Start, Restart) the service.
  - These calls should all fire a request to the service or other function to perform the action. The system waits for acknowledgement of the request but not completion of the action. There is a configurable timeout to wait for each acknowledgement.
- The system collects the responses.
- The system will return a 200 (and empty body) if all action requests are acknowledged. The system will respond with a 500 status and a list of action requests that were not-acknowledged by the timeout as the body of the response.
- If the service does not exist, this can also be included as part of the response.
- The SMA is not considered one of the microservices managed by the SMA. It will not send actions (start, stop, restart) to itself. However, an API needs to be provided on the SMA to stop it.

**Considerations**

- In the future, we would like to have this function be asynchronous and immediately return a token (Memento design pattern) that the client could use to check on the status. However, this would require the SMA to have a state management system, and know how to check the status of each service depending on OS, language, containerization, etc. This is too much for the first iteration.
- It has been determined that the SMA will not worry about service dependencies as these are being resolved with Delhi refactoring and insuring any dependency issues are handled gracefully.

Issue: What do other system management APIs do, what do the specs say (MEC, OpenStack/Windriver (StarlingX), Kubernetes, Docker, sysd, Snappy)?

Config(services []string)

For each service in the services list, collect and return the configuration for each service by making a call to each service for its configuration.

**Implementation**

- For ea: service
  - Make asynchronous call (go routine) to the microservice’s API for configuration.
  - The system waits for the response. There is a configurable timeout to wait for each response.
- The system collects the responses for all requests.
- The system will return a 200 status if all services respond and JSON response per below.
  - [{“service”:“service-name”, “config”:[“property key”:property-value,…]}…]
- Note: the structure of the configuration data will be returned in JSON as it is provided by the micro service. However, the structure will likely be more

hierarchical (versus simple key/value pairs shown above) given the changes being made to configuration for Delhi.

- In the event that the services do not all respond, a 500 status is returned with the body containing the list of services that did not respond.
    - In the case of a service not able to respond but others requested do, the system will still return a 500 status, but it will also include the successful responses from the other services (per above) in the body. An empty response will be recorded for any erroring service.

**Considerations**
The system should respond with JSON and the configuration property keys should follow the naming standards established by Configuration V2. TOML is the configuration file format and is easier for human editing, but JSON is used internally for machine use/readability.

Metric(metrics []string, services []string)

For each service in the services list, collect and return the metric data for each service by making a call to collect the metric for that service.
For the first implementation of the SMA, it is intended that only memory metric will be requested.
Each service should have a configuration setting which indicates which metrics it provides/supports. While initially, all services may return the same metrics, in the future this could be different. Clients (including SMA) can request the configuration setting from each service to know what metrics are supported by that service.
**Implementation**

- For ea: service
    - Make asynchronous call (go routine) to the microservice's API for the metrics.
    - The system waits for the response. There is a configurable timeout to wait for each response.
- The system collects the responses for all requests.
- The system will return a 200 status if all services respond and JSON response per below.
    - [{"service":"service-name", "metrics":["metric name":"value",…]}…]
- In the event that the services do not all respond, a 500 status is returned with the body containing the list of services that did not respond.
    - In the case of a service not able to respond but others requested do, the system will still return a 500 status, but it will also include the successful responses from the other services (per above) in the body. An empty response will be recorded for any erroring service.

**Considerations**
None

## Microservice System Management API

Note: All REST API call body and responses are in JSON.

Each EdgeX micro service must be able to provide the following public REST APIs

- Stop this microservice
    - api/v1/operation (via POST)
    - The body of the request must specify the operation action (with stop the only supported action at this time) and optional parameters (example:  force or graceful).  Example:  { “action”: “stop”, “params”: [“force”:true] }
    - The parameter list is optional and open ended.
    - The API returns 200 (ok) status on successful receipt of the request.
    - Discussion:  Is this a required or optional endpoint – do we want the service to stop itself or do we want the OS to do this?
        - Services would all implement, but SMA may not call on it based on deployment/configuration
        - Required but may not be called depending on what SMA is using.
        - Deployment mechanism would have “smarts” to do it too; and may run counter to internal stop inside microservice
        - SMA should have abstraction to call the “right” stop.  May be this API or it may be a Docker or SNAP or Kubernetes “Stop”.  EdgeX first implementation would call this stop – as long as SMA configuration allows for override this
        - Configuration – do we have on/off config for this API or just allow security disallow – decision:  use security
- Get the configuration settings (aka properties) for this microservice
    - api/v1/config (via GET)
    - The body of the request should be empty.
    - The return result will include the service name and an array of config key/value pairs.  Example: {“service”:“edgex-core-data”, “config”:[“port”:48080, “loggingLevel”:”debug”…]}
    - Note: the structure of the configuration data will be returned in JSON as it is provided by the micro service.  However, the structure will likely be more hierarchical (versus simple key/value pairs shown above) given the changes being made to configuration for Delhi.
    - Issue:  considered Hateos pattern:  [https://spring.io/understanding/HATEOAS](https://spring.io/understanding/HATEOAS)?  See response to this consideration above.
- Get the configuration setting (a single property) for this microservice
    - api/v1/config//[config property name]  (via GET)
        - example:  api/v1/config/port
    - The body of the request should be empty.

    - The return result will include the service name and an array of service:config results. Example: [{“service”:“edgex-core-data”, “config”:[“port”:48080]}, {“service”:“edgex-core-metdata”, “config”:[“port”:48081]}]
    - Note: the structure of the configuration data will be returned in JSON as it is provided by the micro service. However, the structure will likely be more hierarchical (versus simple key/value pairs shown above) given the changes being made to configuration for Delhi.
- Setting the configuration (aka property) for a writable (versus read only property for an EdgeX microservice) is outside the scope of this first release.
- Get metrics this EdgeX microservice(s)
    - api/v1/metrics (via GET)
    - The body of the request must specify the metrics of interest (memory, CPU, …). Example: {“metrics”:[“memory”, “CPU”]}
    - The return result will be an array of metric results and the service name. Example: {“service”:“edgex-core-data”, “metrics”:[“memory”:”34MB”, “CPU”:”3%”]}
    - Discussion: Should this data also be structured hierarchically too – Decision: not enough to worry about yet.
    - Like Operation(Stop), make the use of this optional; allow for SMA to determine where to go to get the metric (again using abstraction and allowing the SMA to get the metric from the OS as an example)
- Getting/setting admin or operational status of microservices is beyond the scope of this first release. It only applies to device services at this time.
- Callbacks (and associated registration/deregistration for changes to status, metrics, configuration, etc. is out of scope for this first release of EdgeX.

## Microservice System Management API Implementation

Implementation assumed in Go and therefore expressed in Go idioms. Gorilla Mux or other router, directs the REST client requests to these methods. A system management package defines these functions (which could also be interfaced like db.go). Ideally, the package is common and used by all services in the same way.

Stop(force string)

This function should return acknowledgement of receipt of request (status 200) immediately.
Initially, this may be a simple call to os.Exit() and may ignore the force/graceful parameter for this first implementation. Eventually we will need a graceful shutdown. Options to use context (https://medium.com/@matryer/make-ctrl-c-cancel-the-context-context-bd006a8ad6ff) and/or something like the designed expressed here http://guzalexander.com/2017/05/31/gracefully-exit-server-in-go.html exist for more graceful exits.
**Considerations**

In the future, tracking of the shutdown request has to be monitored and returned by this or other function.
Issue to eventually be explored: How would this impact the service that is Dockerized (or otherwise containerized)? By killing the executable will Docker detect and also shutdown?

Config(services []string)

Per core and support services, each service loads configuration from the file system or Consul at startup (under Init.go). A call to this function can simply map and return the configuration structure in JSON: {"service":"service-name", "config":["property key":property-value,…]} along with a 200 status

Metrics(metrics []string)

In this first implementation, only a request for memory will be supported. A simple check for memory among the metrics list is made and if it is in the list, then a call to a private memory function is made. All other metrics requests will be ignored by the service.
The system will return a 200 status and JSON response per below.

- {"service":"service-name", "metrics":["metric name":"value",…]}

The function to get the memory stat for Go Lang microservices can use these patterns: https://golangcode.com/print-the-current-memory-usage/ and https://golang.org/pkg/runtime/#MemStats.
**Issues**
Is it ok to allow the Go services to use their internal API to return this? Does this not create problems when Docker or OS is used for other metrics or used for other languages?
**Considerations**
In the future, we may want to capture metrics on some sort of interval at the discretion of the service and even persistent them in some type of store/cache; responding with the stored/cached value on API request.
We will want to provide an abstraction layer for implementation of the collection of data (or any of these functions) so that it allows for different implemenations by 3rd parties or even EdgeX in the future. For example, the 3rd party may provide the version that takes care of persistence or pushes the metric data to some place like Grafana.

## Microservice System Management (SM) Callbacks

Each EdgeX microservice must eventually be able to provide the ability to inform interested clients of changes to its configuration, admin status and memory usage, etc. This is consider out of scope for the first implementation.

## SMA & Security

Note: This (newly-added section) is based on current, forward-looking thinking as the security implementation itself is being tightened.

Regarding how security is being handled—at a high-level—for this release, here is a sketch by way of the salient points we want to keep in mind as we bake security into the project itself. More specifically, on the (important) subject of the authentication and authorization of SMA users:

The general scenario starts with when a user of the SMA issues a request and the JWT token is handed to the SMA.

The SMA will need, for example, the role information encoded inside the JWT token.

1. To enable the SMA to leverage the Security API Gateway (aka Kong) being developed for EdgeX, we anticipate that to be straightforward and a matter of adding the "coordinates" of the SMA to the configuration (TOML) of the Security API Gateway. The general format is as follows:

```
[edgexservices]

  [edgexservices.coredata]

    name = "coredata"

    host = "edgex-core-data"

    port = "48080"

    protocol = "http”

  ...

  ...

  [edgexservices.mgmt]

    name = "mgmt"

    host = "edgex-sys-mgmt"

    port = "48xxx"

    protocol = "http"
```

Further Kong details can be found here. Kong exposes REST endpoints, and some sample requests can be found in the readme page of the project under "Access existing microservice APIs like ping service of command microservice". For example, to use JWT as query string:

- ➢ curl -k -v -H "host: edgex" https://kong-container:8443/command/api/v1/ping?jwt= <JWT from account creation

Alternatively, one can use JWT in the HEADER

- ➢ curl -k -v -H "host: edgex" https://kong-container:8443/command/api/v1/ping -H "Authorization: Bearer <JWT from account creation>"

2. As for the secret store that will be the other important part of the security picture, we will lean on an implementation of a secret store to keep secrets for EdgeX (It is being used by security-api-gateway currently to retrieve the certificate). Please refer to the README of security-api-gateway for the usage.
3. Other than that, the additional information that SMA needs will be just these two: host and port (We leverage constants from there on, once the \V2 code reorganization work—currently under way—has been put in place).

---

Comments from A. Ahmad (Dell) on 8/16

My gut engineering instinct is that the returned result—from a call initiated to api/v1/metrics (via GET)—that contains metrics should *also* (and much like the results from invoking api/v1/config (via GET)) be structured *hierarchically*: That would make interpreting the data easier, for downstream consumption. But I agree with the decision to not worry about it *yet*

---

Comments from T. Conn (Dell) on 7/23

https://wiki.edgexfoundry.org/download/attachments/17498212/System Management Design-v3.odt?version=1&modificationDate=1531871245000&api=v2

I have some thoughts on the API design for the Agent.

First, since each of the stop/start/restart endpoints affect an action, I recommend they be facilitated through either PUT/POST verbs. GET should really be read-only and shouldn't affect underlying state.

The endpoints would take a request body like so:

{ "services": ["edgex-core-data", "edgex-core-metadata", …] }

In this way you would start/stop as many of the services as required using one request, providing a list of service keys in the array. This would eliminate the need for stop/stopall and so forth.

The agent should return some indication of whether the given operation was successful. Perhaps the agent API responds with a token for the above request. That token can be used to fetch status for a given batch. That way callers to the SMA won't be bottlenecked waiting on stubborn processes to either stop/start. When accessing the agent via GET/api/v1/batch/<token> the response could be a K/V structure where K=service key and V=some kind of status (was the op successful).

The agent should also manage its own internal timeouts if services aren't responsive. After a service is either stopped/started, the SMA will need to issue a ping to specified endpoint to verify it was taken down/brought up successfully. The result of that ping goes into the V of the K/V structure above.

Second, w/r/t the supported endpoint of each individual service, recommend that because it's affecting a state change "api/v1/stop" be changed to a PUT/POST against an "api/v1/ops/" endpoint. The body of the request would contain an indication of the action.

Third, the callbacks worry me. We can discuss tomorrow during the call.

---

## Comments from E Attia (Intel) on 7/21

- The semantics of the metric is up to clients to interpret. Maybe we can have a pre-defined set to start (e.g. memory usage) or a set that all services have to cover.

- The frequency of getting these metrics needs to be configurable in the SMA as well.

I thought I send you my notes before going on vacation next week. The only addition that we would like to make is to the metrics API. Currently it has a Get for memory usage. We recommend replacing that API with a more generic API that will return any metric. We have use cases that we are implementing to monetize algorithms. The algorithms are mainly AI for now but that could be anything like water/gas utilities, etc. The algorithms will have many metrics to expose to a web app or any client. These metrics include things like number of inferences, number of objects detected, time per inference, inferences per compute resource, etc. etc. Also, memory and CPU usages are very important among other system resource metrics. So the proposed API is simply 2 parts. One part to get the names of all metrics from the service and another API to get the value for a metric specifying a name from the list returned from the first API. The values could be doubles for now.

We’d prefer to piggy back on the SMA rather creating a new component for metrics.

---