Scalable Microservice Based Architecture For Enabling DMTF Profiles

Divyanand Malavalli
Client Manageability Group
AMD India Pvt
102-103 EPIP, Whitefield, Bangalore, KA 560066 India
Divyanand.Malavalli@amd.com

Sivakumar Sathappan
Client Manageability Group
AMD India Pvt
102-103 EPIP, Whitefield, Bangalore, KA 560066 India
Siva.Sathappan@amd.com

***Abstract*—This paper proposes an architecture for implementing DMTF management Profiles in middleware layer of a management console, using microservices. Microservices is a software architecture style, gaining popularity for developing Internet scale applications. This paper will also provide a comparison with the current prevalent method of design. This paper will also discuss how this microservice can be exposed utilizing REST thus further making it scalable, lightweight etc. This scalable microservice in turn can interact with the managed device using either SOAP (as defined in DASH or SMASH) or REST (as defined in Redfish).**

***Keywords*—Webservices, Microservices, SOA, DMTF Profiles, DASH, REST, JSON, SOAP, XML, Enterprise, Architecture, Manageability**

I. Introduction

DMTF publishes the management profiles, which define the CIM model and its relationship with the management domain. A profile consists of CIM classes, associations, indications and methods that describe the particular management domain. A set of these profiles are defined as specification and released as suite for standardizing the management actions for a particular domain. The suite defines the semantics, protocols and enables interoperability. One such popular suite is 'DASH', which is a standards based management for secure out of band management for desktops and notebooks.

Currently methods exist for providing access to these profiles for developers and end-users to utilize the rich set of information provided by the specification. Within this specification, based on their usage, different users might be interested in a small set of profiles and may not require all the profiles in the suite. Users might be implementing light-weight stack and hence might want to use limited set of profiles. The challenge is to let the users select the profiles they are interested in and provide access only to those profiles.

Each of these profiles defines a vast set of interfaces and developers implementing these profiles would want proceed in a phased manner. Depending on the requirements dictated by the market, certain profiles might be implemented in greater depth. Developers would want to deploy the new additions to the end-users, without disrupting the existing infrastructure either with minimal or zero down-time.

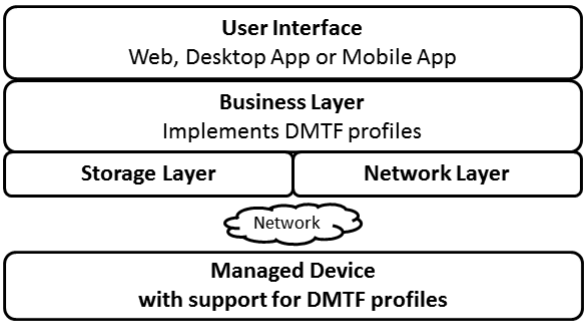
Based on these needs, we are proposing the microservices architecture to overcome the current complexities involved in providing easy access to DMTF profiles.

Terminology:

- Client: The initiator of request. Client is usually the management console. It can also be browser or a mobile app.
- Managed device: Device which implements DMTF profiles and capable of being managed. Device can be desktop computer, notebook, any handheld or fixed device connected to network.

II. Prevalent Architecture

The most common model used as basis for designing management applications is monolithic architecture. This model is based on layered architecture. Each layer is tightly coupled and centrally integrated.



Monolith Design

Figure 1

Typically, 4 distinct layers can be identified in a monolith design apart from the managed device:

978-3-901882-77-7 (c) 2015 IFIP

1) User interface layer
2) Business logic layer
3) Storage layer
4) Network layer

User interface layer presents the interface with which the end user can interact. It takes the input from the user and presents the output results to something the end-user can understand. The UI layer is typically a thick client, which is a desktop client. Web interface based on HTML & JavaScript is also prevalent. In recent times, the user interface is also presented as a mobile app.

Business logic layer occupies the middle layer and implements the domain logic. This layer describes how a particular request must be handled and what must be the response. It coordinates the application, processes commands, makes logical decisions and performs calculations. It also moves and processes data between the two surrounding layers.

In the storage layer, data is stored and retrieved from database. Usually, a single database is used in the implementation.

The network layer enabled communication with managed device over the network. In management applications, data is retrieved from managed device, stored in database and then sent to highest layer for presentation.

The architecture is centralized and the invocation of any functionality in the component of another layer is via a function call.

Typically, all the 4 layers are built together and packaged in a single release. The modules within a layer is developed using the same framework and same programming language. Common observation made is each of these layers represents the project teams in that organization. So the model is based on build, test & release cycles. Every fix or new feature will need a release vehicle. So release cycle is long.

For scaling, entire monolith must be replicated across the servers.

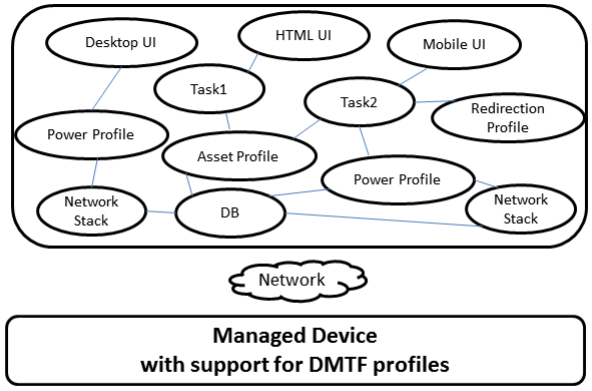
III. Microservices Architecture

Microservices is a software architecture style, where in complex applications are composed of small independent processes (called services). These services communicate with each and also can be invoked via REST APIs. These services are small, highly decoupled and focus on doing a small task. So a microservice design consists of a suite of independent deployable services.

The RESTful architectural style must consist of these constraints:

1. Client-Server: Clients and servers are separated via an interface. This separation means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.
2. Stateless: The necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. So the server need not maintain, update or communicate that session state.
3. Cacheable: Responses must define themselves as cacheable, or not, to prevent clients reusing stale or inappropriate data in response to further requests.
4. Uniform Interface: The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the architecture, which enables each part to evolve independently.
5. Layered System: A client need not know whether it is connected to an end server or to an intermediary. Intermediary servers improve system scalability by enabling load-balancing.

A typical microservices architecture style based RESTful interfaces is shown in figure 2:



Microservices Design

Figure 2

Each of these small services run within their own process and communicate with HTTP API. Clear constructs and clear agreements are defined and established between these services. So any programming language can be used for developing these services as long as they adhere to the agreed protocol. Hence a microservice design can consists of services developed with different languages and different frameworks. So the best tool can be used for developing any service. Every service can have its own build cycle and can be deployed independent of each other. Each component is a service. Each component is complete in itself. The component is usually developed by

cross-functional teams. The development is done by product teams, which own and run the services.

Each service performs a business capability. Microservice is based on these principles,

- Each service does one thing well
- Portability is preferred over efficiency

Services are the end points and are smart. The pipe connecting these services is assumed to be dumb. The invocation of any service is via HTTP request APIs and response is also via HTTP, response APIs. Hence messaging communication framework is lightweight. The invocation of any functionality in the component of another layer is via a HTTP call.

The architecture is de-centralized and data can be stored in one or more databases. The database micro service will abstract all the database operations from other services.

The architecture enables the paradigm of continuous delivery & rapid deployment. It provides a mechanism to handle elasticity to demand. Latest tools can be employed for building the services. New concepts can be implemented and showcased without affected the existing functionality.

Failure of one or more components won't bring down the whole system. Failure of one service means the system won't offer that particular service. The rest of the system works normally.

For monitoring all the services in the system, a dashboard is required.

Scaling is simple. It can be done in 2 ways. Within the same server, multiple instances of the service in demand can be instantiated to handle additional requests. Other way is to distribute these services across servers. Since the messaging is based on HTTP, scaling is transparent to requesting entities.

IV. Proposed Architecture

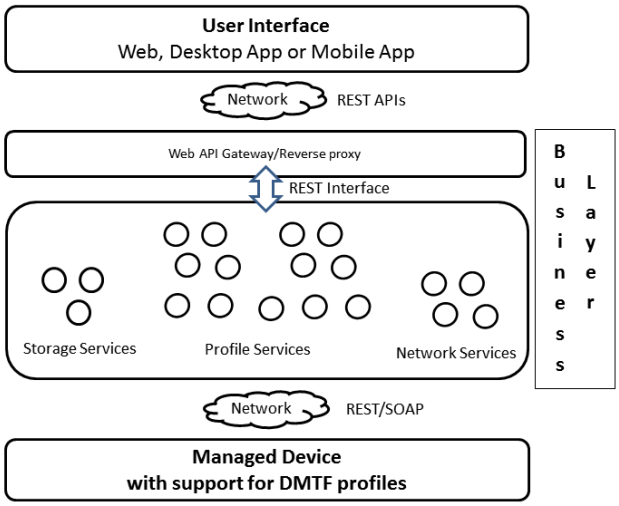
The concept of microservices is adapted in the business logic layer for implementing DMTF profiles in management application. The proposal for middle tier, business logic layer consists of 2 distinct layers:

1. Web API Interface Layer
2. Services Layer

Internally, the web API interface layer consists of 2 parts - Web layer and API Gateway layer. The services layer consists of different microservices which address different requests. Some of these microservices implement DMTF profiles, some network communication stack and some storage points.

The user interface raises a request based on REST API. The API interface layer would interpret this request and proxy the request to one or more microservices. Each microservice would fetch data from either storage or fetch via network from managed device. The result data would be sent back as response to user interface in the format it requested.

The key idea is illustrated in the diagram below.



Proposed design for enabling DMTF profiles

Figure 3

a) The web layer provides the HTTP interface and acts as load balancer. It sends the request to API gateway layer and later sends the response back to the original requester. Depending on the implementation, if there are multiple API gateways, the web layer acts as reverse proxy and proxies the request to a specified API gateway, and gets the response.

b) The API gateway layer is the single entry point for all requests. Depending on the request, it either sends the request to the appropriate microservice or it fans out multiple microservices to handle the request. This layer insulates the clients from how the application is partitioned into microservices. Though this layer adds complexity to the application, it provides faster service by reducing the number of requests from client to server. It simplifies the client by moving logic for calling multiple services from the client to API gateway.

c) Services layer consists of 3 types of microservices,

- DMTF profiles services
  These services implement the various profiles as per DMTF specification. These services typically format the request as per managed device. They decode the response from managed device and sends back to API gateway layer for customizing the response as per client.

- Network connectivity services
  These services provide end point connectivity to managed devices. Typical connectivity is over HTTP. It also provides secure connectivity. Depending on the implementation of the managed device, this layer can provide both SOAP & REST services. Hence can connect to both legacy managed devices implemented on SOAP interfaces and newer managed devices implemented on REST interfaces.

- Data storage services
  Certain data points might be required to be stored for later access. The data services layer provides the storage functionality. Typically time-consuming tasks are performed in the background and the respective services update the results in the database with the help of data services. Depending on the quantity of data generated either SQL or NoSQL or both databases can be supported in this architecture. Also, wide column databases with as Hadoop, big data can also be supported. That would depended on the storage and retrieval demands of the management console.

V. USAGE FLOW

The management service developer would implement and publish the REST APIs for consumption. The REST API would internally access DMTF profile implementation. So the REST API is facade for DMTF profiles.

A management console developer would utilize these REST API services and provide a mechanism for the end user to communicate with the managed device.

Typically, if an user wants to get the power status of a managed device, that user would call the REST API defined for power on that device. The REST API framework would fetch the previously stored authentication details and query the managed device for power status using DMTF profile specification.

VI. BENEFITS OF REST APIS

SOAP & REST both provide a mechanism of non-binary messaging framework and rely on well-established rules for exchanging information. But there are certain inherent differences.

SOAP is the older of the two and relies exclusively on XML to provide messaging services. SOAP has been around for some time and has been standardized. Hence it is extended to work on wide variety of network topology and configurations. SOAP is highly extensible and has built-in error handling. Another aspect of SOAP is that it can be used on any transport, not just limited to HTTP. The XML used to make requests and receive responses in SOAP can become complex. The libraries used for generating XMLs can be bulky and may not be the choice for light weight applications.

REST provides a lighter weight alternative to SOAP. Instead of using XML to make a request, REST relies on a simple HTTP URL. REST can use any of four different HTTP 1.1 verbs (GET, POST, PUT, and DELETE) to perform tasks. REST-based web services typically output the data in JavaScript Object Notation (JSON) format. JSON is prevalent but any other format can be used depending in the discretion of the designer.

Either SOAP or REST, each has its own definite advantages and disadvantages. To summarize, SOAP is standardized, works well in distributed enterprise environments, provides significant pre-build extensibility in the form of the WS* standards and has built-in error handling. REST is lightweight, fast, easier to use and has smaller learning curve. Messaging scheme is efficient in case of REST. Select between SOAP and REST is to be made based on the programming language, development framework, the environment in which the application will be deployed, and the requirements of the application. Either can be a better choice depending on the problem domain.

In the architecture we have proposed, both SOAP & REST based managed devices can be managed, co-exist and work independent of each other. Some of the microservices can use SOAP, while others can use REST and communicate with the managed device. This approach will provide a mechanism to address existing SOAP deployments and also to communicate with newer REST based implementations.

VII. BENEFITS OF THE PROPOSED ARCHITECTURE DESIGN

The flexibility and usefulness of the proposed architecture design is outlined.

1. Easy to showcase, demo new profiles in existing framework
2. Enhance and deploy any existing profile, without affecting other profiles. This also reduces testing effort.
3. Microservices framework provides scaling functionality
4. Mix of services implemented in different languages can co-exist.
5. Support both SOAP and REST based managed devices.
6. Approach is market centric. Based on the requirements dictated by the market, a given component can be modified and deployed independent of this system.

VIII. POTENTIAL PITFALLS

The microservices framework, which is based on distributed computing, if not designed properly suffers from these issues.

- Network reliability: The end points are connected over network and so the network must be reliable.

- Latency: The path between end-points may span multiple networks with heterogeneous topology. The solution must consider the latency aspect.
- Bandwidth: The communication from client, between microservices and to managed device utilizes the same network. So network traffic might be high. The implementation must consider bandwidth availability during normal & peak operation.
- Network security: The communication channel is based on open protocols and hence adequate checkpoints & guards must be implemented to keep away malicious content in the network.
- Interface definition: The communication between microservices and with external world must be strict and well defined. As the system evolves this definition might become complex.
- User authorization: By default the APIs of microservices are open to all authenticated users. This may be designed in hierarchical setup, which is typically the case in enterprise management. User's permissions must be checked and their role analyzed before providing access the microservices.
- Setting up microservices: Initial setup of microservices components might require more effort than similar monolith setup. But adding additional features to existing microservices setup will be simpler.

IX. Implementation

A typical microservices implementation design consists of a web server and a REST API library. The web server handles the HTTP requests and an option handle TLS connection. The API specification can be defined with the REST API library component. There are many open source and closed source both free and paid options available for web server. For REST API library, most modern languages and frameworks have one or more options.

For implementing DMTF profiles, the profile SDK is required. There are both free and paid options available. For instance, for implementing DMTF's DASH specification, AMD's DASH SDK can be used.

The REST API specification and DMTF profile specification implementation SDK are tied, held and employed together by the management console. There is no generic management console software available. All management consoles are built to purpose. Depending on the problem domain, the management console is built to address that problem domain.

The scope of this paper doesn't cover the practical implementation or the proof of concept. The individual technologies employed in the proposed architecture are well established and this paper tries to bring out the plan to expose DMTF profiles to the end users in a scalable method.

X. Conclusion

In this paper, we have proposed an alternate approach for supporting DMTF profiles in commercial management consoles. We have looked at traditional architectural design methodology of management consoles. We have proposed microservices architecture for designing the middle tier, or the business layer which implement the domain logic in management consoles.

Figures

[1] Figure 1: Depiction of monolith design
[2] Figure 2: Demonstration of the concept of microservices
[3] Figure 3: Proposed architecture for implementing DMTF profiles.

References

[1] DMTF Management Profiles, Available: http://www.dmtf.org/standards/profiles, [Accessed: May 2015].

[2] Roy Fielding, "Architectural Styles and the Design of Network-based Software Architectures", [Doctoral Dissertation] Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, [Accessed: May 2015].

[3] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", http://research.microsoft.com/en-us/um/people/lamport/pubs/implementation.pdf [Accessed: September 2015].

[4] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", 1982 (available) http://www.cs.cornell.edu/courses/cs614/2004sp/papers/lsp82.pdf [Accessed: September 2015].

[5] Martin Fowler, "Microservices", [Blog entry] Available: http://martinfowler.com/articles/microservices.html, [Accessed: May 2015].

[6] E. A. Brewer, "Towards Robust Distributed Systems", 2000 http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf [Accessed: September 2015].

[7] E. Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed", 2012, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed [Accessed: September 2015].

[8] John Mueller, "Understanding SOAP and REST Basics" [Blog entry] Available: http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/, [Accessed: May 2015].

[9] Stubbs, J., Moreira, W., Dooley, R., "Distributed Systems of Microservices Using Docker and Serfnode", Science Gateways (IWSG), 2015 7th International Workshop, [Accessed: September 2015].

[10] Vianden, M., Lichter, H., Steffens, A., "Experience on a Microservice-Based Reference Architecture for Measurement Systems", Software Engineering Conference (APSEC), 2014 21st Asia-Pacific, [Accessed: September 2015].

[11] Fred George, "Micro Services Architecture", YOW! 2012 https://yow.eventer.com/yow-2012-1012/micro-services-architecture-by-fred-george-1286 [Accessed: September 2015].

[12] Review of DMTF DASH applications and their architecture, Available: www.amd.com/DASH, [Accessed: April 2015].

[13] Microservices Patterns, Available: http://microservices.io/patterns/ [Accessed: May 2015].