UNIT 9: DEVELOPING AN API FOR A DISTRIBUTED ENVIRONMENT
&
UNIT 10: FROM DISTRIBUTED COMPUTING TO MICROARCHITECTURES

NOTES

Unit 9: Developing an API for a distributed environment

Pillai, A.B. (2017) Software Architecture with Python. Birmingham, UK. Packt Publishing Ltd.Chapter 5.

- When the system scales by either adding or making better use of resources inside a compute node, such as CPU or RAM, it is said to scale vertically or scale up. On the other hand, when a system scales by adding more compute nodes to it, such as a creating a load-balanced cluster of servers, it is said to scale horizontally or scale out.
- The degree to which a software system is able to scale when compute resources are added is called its **scalability**. Scalability is measured in terms of how much the system's performance characteristics, such as throughput or latency, improve with respect to the addition of resources. For example, if a system doubles its capacity by doubling the number of servers, it is scaling linearly.
- **Concurrency** is the amount of work that gets done simultaneously in a system.

We find that there is a relation between Scalability, Performance, Concurrency, and Latency. This can be explained as follows:

1. When performance of one of the components in a system goes up, generally the performance of the overall system goes up.
2. When an application scales in a single machine by increasing its concurrency, it has the potential to improve performance, and hence, the net scalability of the system in deployment.
3. When a system reduces its performance time, or its latency, at the server, it positively contributes to scalability.

We have captured these relationships in the following table:

Concurrency	**Latency**	**Performance**	**Scalability**
High	Low	High	High
High	High	Variable	Variable
Low	High	Poor	Poor

An ideal system is one that has good concurrency and low latency; such a system has high performance, and would respond better to scaling up and/or scaling out.

A system with high concurrency, but also high latency, would have variable characteristics—its performance, and hence, scalability would be potentially very sensitive to other factors such as current system load, network congestion, geographical distribution of compute resources and requests, and so on.

Concurrency

A system's concurrency is the degree to which the system is able to perform work simultaneously instead of sequentially. An application written to be concurrent in general, can execute more units of work in a given time than one which is written to be sequential or serial.

When one makes a serial application concurrent, one makes the application better utilize the existing compute resources in the system—CPU and/or RAM—at a given time. Concurrency, in other words, is the cheapest way of making an application scale inside a machine in terms of the cost of compute resources.

Concurrency can be achieved using different techniques. The common ones include the following:

1. **Multithreading**: The simplest form of concurrency is to rewrite the application to perform parallel tasks in different threads. A thread is the simplest sequence of programming instructions that can be performed by a CPU. A program can consist of any number of threads. By distributing tasks to multiple threads, a program can execute more work simultaneously. All threads run inside the same process.
2. **Multiprocessing**: Another way to concurrently scale up a program is to run it in multiple processes instead of a single process. Multiprocessing involves more overhead than multithreading in terms of message passing and shared memory. However, programs that perform a lot of CPU-intensive computations can benefit more from multiple processes than multiple threads.
3. **Asynchronous Processing**: In this technique, operations are performed asynchronously with no specific ordering of tasks with respect to time. Asynchronous processing usually picks tasks from a queue of tasks, and schedules them to execute at a future time, often receiving the results in callback functions or special future objects. Asynchronous processing usually happens in a single thread.

Python, especially Python 3, has built-in support for all these types of concurrent computing techniques in its standard library. For example, it supports multi-threading via its *threading* module, and multiple processes via its *multiprocessing* module. Asynchronous execution support is available via the *asyncio* module. A form of concurrent processing that combines asynchronous execution with threads and processes is available via the *concurrent.futures* module.

In the coming sections we will take a look at each of these in turn with sufficient examples.

NOTE: The asyncio module is available only in Python 3

Both concurrency and parallelism are about executing work simultaneously rather than sequentially. However, in concurrency, the two tasks need not be executed at the exact same time; instead, they just need to be scheduled to be executed simultaneously. Parallelism, on the other hand, requires that both the tasks execute together at a given moment in time.

> However, on a multi-core CPU, two threads can perform parallel computations at any given time in its different cores. This is true parallelism.
>
> Parallel computation requires that the computation resources increase at least linearly with respect to its scale. Concurrent computation can be achieved by using the techniques of multitasking, where work is scheduled and executed in batches, making better use of existing resources.

Resource constraint – semaphore versus lock

We saw two competing versions of implementing a fixed resource constraint in the previous two examples—one using `Lock` and another using `Semaphore`.

The differences between the two versions are as follows:

1. The version using Lock protects all the code that modifies the resource—in this case, checking the counter, saving the thumbnail, and incrementing the counter—to make sure that there are no data inconsistencies.
2. The Semaphore version is implemented more like a gate—a door that is open while the count is below the limit, and through which any number of threads can pass, and that only closes when the limit is reached. In other words, it doesn't mutually exclude threads from calling the thumbnail saving function.

Hence, the effect is that the Semaphore version would be faster than the version using Lock.

Concurrency options – how to choose?

We are at the end of our discussion of concurrency techniques in Python. We discussed threads, processes, asynchronous I/O, and concurrent futures. Naturally, a question arises—when to pick what?

This question has been already answered for the choice between threads and processes, where the decision is mostly influenced by the GIL.

Here are somewhat rough guidelines for picking your concurrency options.

- **Concurrent futures vs Multi-processing:** Concurrent futures provide an elegant way to parallelize your tasks using either a thread or process pool executor. Hence, it is ideal if the underlying application has similar scalability metrics with either threads or processes, since it's very easy to switch from one to the other as we've seen in a previous example. Concurrent futures can be chosen also when the result of the operation needn't be immediately available. Concurrent futures is a good option when the data can be finely parallelized and the operation can be executed asynchronously, and when the operations involve simple callables without requiring complex synchronization techniques.

  Multi-processing should be chosen if the concurrent execution is more complex, and not just based on data parallelism, but has aspects like synchronization, shared memory, and so on. For example, if the program requires processes, synchronization primitives, and IPC, the only way to truly scale up then is to write a concurrent program using the primitives provided by the multiprocessing module.

  Similarly when your muti-threaded logic involves simple parallelization of data across multiple tasks, one can choose concurrent futures with a thread pool. However if there is a lot of shared state to be managed with complex thread synchronization objects – one has to use thread objects and switch to multiple threads using `threading` module to get finer control of the state.

- **Asynchronous I/O vs Threaded concurrency:** When your program doesn't need true concurrency (parallelism), but is dependent more on asynchronous processing and callbacks, then `asyncio` is the way to go. Asyncio is a good choice when there are lot of waits or sleep cycles involved in the application, such as waiting for user input, waiting for I/O, and so on, and one needs to take advantage of such wait or sleep times by yielding to other tasks via co-routines. Asyncio is not suitable for CPU-heavy concurrent processing, or for tasks involving true data parallelism.

Scalability architectures

As discussed, a system can scale vertically, or horizontally, or both. In this section, we will briefly look at a few of the architectures that an architect can choose from when deploying his systems to production to take advantage of the scalability options.

Vertical scalability architectures

Vertical scalability techniques comes in the following two flavors:

- **Adding more resources to an existing system**: This could mean adding more RAM to a physical or virtual machine, adding more vCPUs to a virtual machine or VPS, and so on. However, none of these options are dynamic, as they require stopping, reconfiguring, and restarting the instance.
- **Making better use of existing resources in the system**: We have spent a lot of this chapter discussing this approach. This is when an application is rewritten to make use of the existing resources, such as multiple CPU cores, more effectively by concurrency techniques such as threading, multiple processes, and/or asynchronous processing. This approach scales dynamically, since no new resource is added to the system, and hence, there is no need for a stop/start.

Horizontal scalability architectures

Horizontal scalability involves a number of techniques that an architect can add to his tool box, and pick and choose from. They include the ones listed next:

- **Active redundancy**: This is the simplest technique of scaling out, which involves adding multiple, homogenous processing nodes to a system typically fronted with a load balancer. This is a common practice for scaling out web application server deployments. Multiple nodes make sure that an even if one or a few of the systems fail, the remaining systems continue to carry out request processing, ensuring no downtime for your application.

  In a redundant system, all the nodes are actively in operation, though only one or a few of them may be responding to requests at a specific time.

- **Hot standby**: A hot standby (hot spare) is a technique used to switch to a system that is ready to server requests, but is not active till the moment the main system go down. A hot spare is in many ways exactly similar to the main node(s) that is serving the application. In the event of a critical failure, the load balancer is configured to switch to the hot spare.

Summary

In this chapter, we reused a lot of ideas and concepts that you learned in the previous chapter on performance.

We started with a definition of scalability, and looked at its relation with other aspects like concurrency, latency, and performance. We briefly compared and contrasted concurrency and its close cousin parallelism.

We then went on to discuss various concurrency techniques in Python with detailed examples and performance comparisons. We used a thumbnail generator with random URLs from the Web as an example to illustrate the various techniques of implementing concurrency using multi-threading in Python. You also learned and saw an example of the producer/consumer pattern, and using a couple of examples, learned how to implement resource constraints and limits using synchronization primitives.

Next we discussed how to scale applications using multi-processing and saw a couple of examples using the `multiprocessing` module – such as a primality checker which showed us the effects of `GIL` on multiple threads in Python and a disk file sorting program which showed the limits of multi-processing when it comes to scaling programs using a lot disk I/O .

We looked at asynchronous processing as the next technique of concurrency. We saw a generator based co-operative multitasking scheduler and also its counterpart using `asyncio`. We saw couple of examples using asyncio and learned how to perform URL fetches using the aiohttp module asynchronously. The section on concurrent processing compared and contrasted concurrent futures with other options on concurrency in Python while sketching out a couple of examples.

We used Mandelbrot fractals as an example to show how to implement data parallel programs and showed an example of using `PyMP` to scale a mandelbrot fractal program across multiple processes and hence multiple cores.

Next we went on to discuss how to scale your programs out on the Web. We briefly discussed the theoretical aspect of message queues and task queues. We looked at celery, the Python task queue library, and rewrote the Mandelbrot program to scale using celery workers, and did performance comparisons.

WSGI, Python's way of serving web applications over web servers, was the next topic of discussion. We discussed the WSGI specification, and compared and contrasted two popular WSGI middleware, namely, uWSGI and Gunicorn.

Towards the end of the chapter, we discussed scalability architectures, and looked at the different options of scaling vertically and horizontally on the Web. We also discussed at some best practices an architect should follow while designing, implementing, and deploying distributed applications on the web for achieving high scalability.

In the next chapter, we discuss the aspect of Security in software architecture and discuss aspects of security the architect should be aware of and strategies for making your applications secure.

Salah, T., Zemerly, M. J., et al. (2016) The Evolution of Distributed Systems towards Microservices Architectures, in Proc. of the 11th International Conference for Internet Technology and Secured Transactions.

EVOLUTION OF SYSTEMS ARCHITECTURES: from basic architecture (client-server), mobile agents, SOA to Microservices

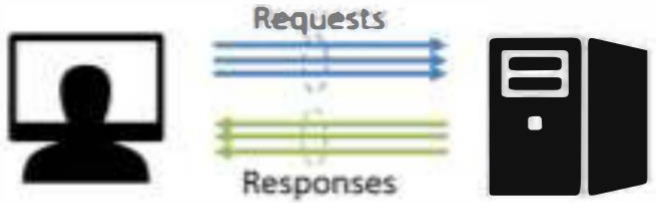

Figure 2. Basic parts and functions of Client-Server Paradigm

B. Drawbacks

Although, client-server model can be flexible, robust, and scalable [13], it yet suffers from many issues [7]:

- Scalability takes time: To scale the service available on the server side, a whole server machine needs to be added in order for the service to scale.
- The way this model scales will raise another disadvantage, which is the hardware cost.
- Frequent updates are hard to maintain: Mostly maintenance is done by the administrator side and it is difficult to distribute the maintenance duty among different personnel.
- Procedure calls done by the clients can affect the load on the network.
- The inability to integrate autonomous services as a whole to be requested by various clients (Supported by SOA architecture - see Section IV).
- Services can be hijacked by enormous number of requests leading to the service to be unavailable.

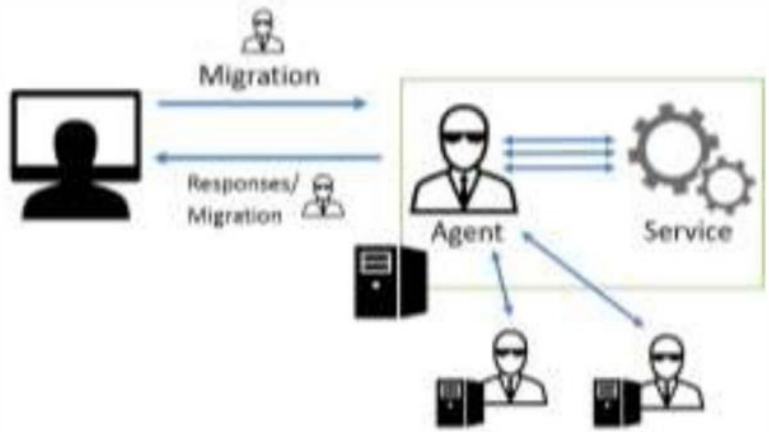

Figure 3. Basic parts and features of Mobile Agents architecture

B. Drawbacks

Mobile agent paradigm was proposed as a viable solution to overcome client-server paradigm limitations as seen in the related work (Part A). Unfortunately, they did not meet the level of expectations they were made for about a decade ago due to the following reasons:

- They were intended to be used over slow networks [5], and often getting offline. Nowadays, facing such problems is avoidable relying on advanced networking methods.
- They were made for client customization purposes; it became possible to add such features in remote calls.
- It requires to take into account major security measures as trusted host, trusted MA community, secure communication protocols and secure MA (to not expose owner profile).
- Not being able to scale; acceptance of executable code hosting and the need for common execution language.
- Hosts should have mobile agent hosting capability which is not widely spread among many machines.
- Agent classes need to be installed on the machines [5]
- Available bandwidth is no longer an issue compared to the need of fast response and execution time
- In [18], Agents are known for their intelligence and autonomous behavior. They can contribute in decision making such as in stock market and trading decisions, but having this agent mobile, is not a necessity anymore.

IV. SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) is another evolution of Client Server model and said to be the most successful. SOA allows services to be loosely coupled, reusable and dynamically assembled which supports the changing business environment [5]. SOA supports explicit boundaries between autonomous services located on different servers to fulfil application requirements. SOA came in order to overcome the challenges of having a large monolithic applications and aims at enhancing the reusability of a service by multiple end-user application [19]. Figure 4 illustrates the basic structure of SOA in an example.

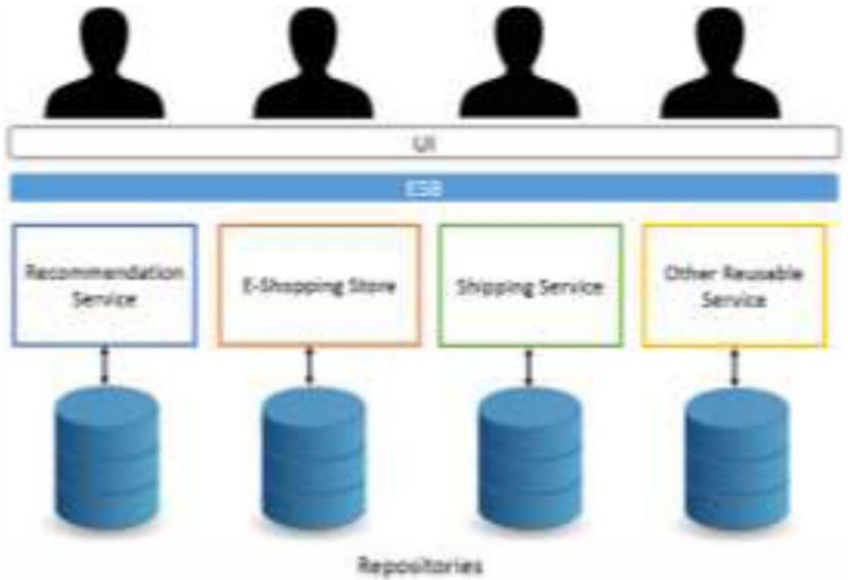

Figure 4. Illustrative example of SOA showing layers and parts relied on

SOA is usually referred to as a business concept enforcing standards among different ownership domains to utilize distributed capabilities [21]. SOA was compared to three alternative architectures [21]: Application Integration, Workflow Integration and Desktop Integration. SOA was concluded to offer improved efficiency and changeability features. SOA can also be used for the purpose of wrapping up loosely coupled web services [22]. Services should share a common standard in order to conduct interaction. In [22], the various SOA protocol stacks are represented in different six layers: Transport, Message, Description, Management, Assemble and Presentation layers. SOA is usually relied on in huge enterprise applications including social media, e-shopping systems, e-banking, and many other online services dealing with a high number of continuous requests. For

B. Drawbacks

The monolith representation of the reusable services seen in the SOA architecture was found not being able to keep pace of the customers and business expectations [19]. In this modern time, developing software with more features in less time and developing scalable solutions with fewer resources are highly demanded. Authors in [25] claimed that although monolithic architectures can contain several services internally, but they have to be deployed as one unit. Several copies of the same application can be run in order to maintain scalability, but they will be identical. This type of architecture is efficient to use for large applications which will make it easy to develop and deploy. The main issue that will rise of such applications is the difficulty for modifying and understanding it. Big applications contain large code base, adding updates or replacing developers for that specific task will be exhaustive and nearly impossible for some applications. Redeploying the whole application might be required for even small components to be added. The solution of running multiple copies for scaling is not efficient as it will result in the increase of memory consumption. In addition, the reusability of common data between the copies as caching solutions will not work. SOA typically centralizes the sophisticated operations including business constraints, message routing, and service orchestration all in the ESB [26]. Modern efforts call for the need of dumb pipes (ESB) and smart endpoints [26]. Due to the illustrated drawbacks of SOA architecture, Micro-services architecture has emerged and grabbed a lot of researchers' attention recently. Microservices architecture is discussed in details in Section V.

V. MICROSERVICES

Users nowadays expect more interactive enrichment along with dynamic user experience on a wide range of devices used by clients. Developer and organizations want to have the ability for frequent updates on their services. Accordingly, relying on monolithic applications is no longer adequate [27]. An alternative architecture proposed by researchers to fulfil modern needs is known as Microservices Architecture [28]. The aim of this architecture is to break the application into a set of smaller independent services [29]. This architecture is often compared with the SOA architecture we came across in Section IV. The most serious drawback that called the need for enhancing SOA is its centralized integration [30]. Some prefer referring to microservices as an approach to build SOA. Others stated that SOA architecture can be referred to as the monolithic representation of an application or as a compressed view of microservices. Figure 5 gives an example of an approach taken to represent microservices architecture relying on the system represented using SOA in Figure 4.

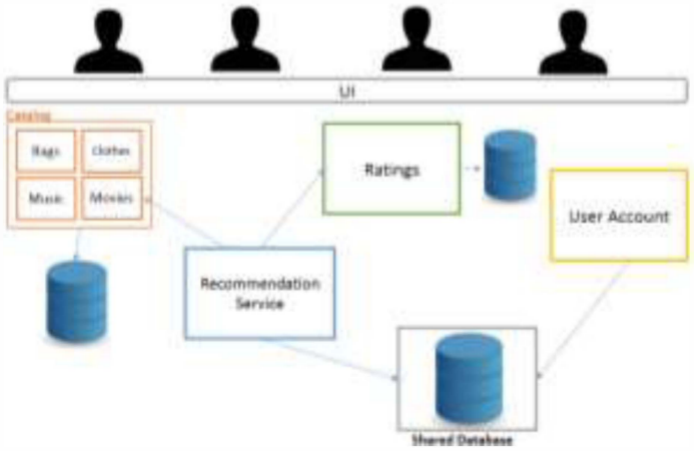

Figure 5. Illustrative example describing the evolved SOA architecture parts to microservices

Table 2. Comparison between distributed systems architectures

Feature	Architecture			
	Client/ Server	*Mobile Agents*	*SOA*	*Microservices*
Network Utilization	No	Yes	Yes	Yes
Inter-process communication	No	Yes	No	Yes
Elastic Scaling	No	No	No	Yes
Mostly Lightweight Communication	No	No	Yes	Yes
Resilience	No	No	Yes	Yes
Fast Software Delivery	No	No	Yes	Yes
Autonomous Service integration	No	No	Yes	Yes
Portable Services	No	Yes	Yes	Yes
Reusability of Services	No	No	Yes	Yes
Service Migration	No	Yes	No	Yes
Monolithic services	Yes	Yes	Yes	No
Loose-coupling	No	No	Yes	Yes
Mostly Cloud-based	No	No	No	Yes
Cross-functional development	No	No	Yes	Yes
Functional Separation	No	No	Yes	Yes
Require Middleware	No	No	Yes	No
High Fault-tolerance	No	No	No	Yes
decentralized governance	No	No	No	Yes
Low Resources Cost	No	No	No	Yes
Independent service deployment	No	No	No	Yes

Winkfield, L., Hu, Y-H., Hoppa, & M. A. (2018) A Study of the Evolution of Secure Software Development Architectures, Journal of the Colloquium for Information System Security Education.

Timeframe	Architecture paradigm	Security / Deployment Concerns	Environment Effects
1970-1990	Monolithic terminal server systems (mainframes)	Deploy once; secure the monolith	Transactional enterprises (e.g. banks, airlines, hospitality)
1990's	Client-server	More nodes to secure increases the security burden	Application logic abstractions; rich user interfaces
Early 2000's	Thin client	HTML forms introduce new vulnerabilities	High-latency; poor performance of JavaScript; browser incompatibilities
Mid-2000's	SOA, web toolkits	W3C standards improved security; but XML requires	Single-sign on is an infrequently realized ideal
		special security measures	
2010	Purely JavaScript frameworks	Rich, responsive, client-side experiences interacting with server-side data via RESTful backends	Strong testing frameworks & improved JavaScript makes security more manageable
Present day	Micro-services, containers	Full-stack developers; demand for client applications and real-time data	Ubiquitous internet, mobile devices, IoT

Table 1: Some Significant Paradigm Shifts in Software Development.

Microservices is a subset of SOAs [21] that describes "a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal" [16]. Microservices applications offer several developmental, operational and security advantages. For example, developers are able to create and deploy smaller incremental features or application changes, without the burden of redeploying an entire application system.

5.1 Modern Software Development Solutions

With speed and agility being paramount to success, naturally solutions have arisen to pursue these advantages. For example, some businesses that previously operated their own on-premise data centers are now opting to partially or fully outsource to cloud IaaS providers, so they can focus their attention on application delivery. Among the platforms being used by businesses to scale and grow are Amazon Web Services [23], Google Cloud Platform [24], and Microsoft Azure [25].

Goals such as speed, agility, flexibility and portability often don't align well with traditional architecture patterns, organizational structures and delivery cadences. To meet the speed of contemporary market demands and therefore remain competitive, companies are adopting delivery and cultural philosophies such as Agile Development and DevOps. Agile Development has taken the traditional waterfall process and reduced the time-to-value by implementing iterative, compressed delivery cycles. This results in shorter feedback loops, reduced risk, fast feature delivery, and greater overall flexibility. DevOps attempts to align the competing goals of developers who want to innovate and move software and products to the market faster, and operations who prefer to keep things stable, so the two can join forces and deliver greater value downstream.

Furthermore, Extensible Markup Language (XML) – which is used to format some SOA application data – is "inherently insecure," and as such special measures must be taken to properly manage its vulnerabilities [12]. Nevertheless, enterprise SOA implementations are prevalent, even today. It wasn't until 2010 that sophisticated, purely JavaScript frameworks such as Ember.JS, Backbone.JS, and Angular.JS [34] [35] revolutionized the modern web by creating rich, responsive, client-side experiences interacting with server-side data via RESTful backend services [35].

Unit 10: From Distributed Computing to Microarchitectures

Arnaut, W., Oliveira, K. & Lima, F. (2010) OWL-SOA: A Service Oriented Architecture Ontology Useful during Development Time and Independent from Implementation Time, IEEE.

- service oriented architecture (SOA)
- Ontologies have been used to organize services in repositories, as proposed by of OWL-S (Ontology Web Service Language) and WSMO (Web Service Modeling Ontology). Meanwhile, these ontologies have two broad restrictions: they are used on runtime only, and deal only with Web Service technology for the implementation of services.

Different technologies are used to categorize services offering support to search and recovery in an SOA repository, as for example: protocols such as XML (eXtensible Markup Language) and XML Schema, UDDI (Universal Description, Discovery and Integration), the RAS standard (Reusable Asset Specification[13], and ontologies. However, the protocols XML focus only on syntactic and structural aspects, lacking information on the semantic level, and the UDDI merely describes the functionalities of the services registered in a language that is incomprehensible to machines[14].

The RAS standard defines a model for the representation of assets produced throughout the software development process [13]in such a way as to allow its reuse, being amply utilized to describe and indicate software components. Despite its wide use, the RAS standard does not add semantics to its search and recovery of services.

B. Ontologies in SOA

Ontologies are explicit formal specifications of terms in a domain and their relationship to other terms [15], containing a set of concepts (entity, objects, domains, processes, goals, and results), properties, relationships, restrictions, or axioms [16].

Munir, K. & Sheraz Anjum, M. (2018) The use of ontologies for effective knowledge modelling and information retrieval, Applied Computing and Informatics. 14: 2. 116-126. DOI: https://doi.org/10.1016/j.aci.2017.07.003.

- The dramatic increase in the use of knowledge discovery applications requires end users to write complex database search requests to retrieve information. Such users are not only expected to grasp the structural complexity of complex databases but also the semantic relationships between data stored in databases. In order to overcome such diculties, researchers have been focusing on knowledge representation and interactive query generation through ontologies, with particular emphasis on improving the interface between data and search requests in order to bring the result sets closer to users research requirements. This paper discusses ontology-based information retrieval approaches and techniques by taking into consideration the aspects of ontology modelling, processing and the translation of ontological knowledge into database search requests. It also extensively compares the existing ontology-to-database transformation and mapping approaches in terms of loss of data and semantics, structural mapping and domain knowledge applicability. T

- In information management systems, structured query formulation languages are one means of retrieving information. Writing structured queries is a powerful method to access data since it allows end-users to formulate complex database queries by learning specialised query languages. However, query formulation with the exception of a few visual query generation and refinement approaches remains appreciatively dicult for the various levels of systems users. In recent years information retrieval has turned out to be more complicated with the increased use of data mining, decision support and business analytics applications.

- Recently, semantic-based approaches using domain ontologies have been adapted for data modelling and information retrieval. Ontologybased information retrieval, for example as in [4], [5] and [6] mainly aim at improving the interface between data and search requests in order to bring the result sets closer to the users' research requirements.

Ontology languages

- Numerous ontology languages were developed in the last few years. Most of these are based on the eXtensible Markup Language (XML) [9] which enables them to be machine interpretable [10]. Notable examples are the Resource Description Framework (RDF) and RDF Schema [11], the DARPA Agent Markup Language and the Ontology Inference Layer (DAML+OIL) [12], and the Ontology Web language (OWL) [13] and OWL2 [14].

- The search method of ontology-based image retrieval systems such as Ontogator [23], and OntoViews [24] are examples of a concept-based multi-facet search using RDFS ontologies. In a multi-facet search, multiple distinct views are augmented to data created via ontology projection[24]. OntoViews, supports semantic auto-completion of a query [24]. It uses a keyword search mechanism for ontology navigation. The search keywords are linked directly to ontology classes. A user search request is processed as a multi-facet search and results are delivered in a web browser. Once a single interesting instance (at least) has been found, additional information can be retrieved via ontology browsing.

- Effective information retrieval is becoming more challenging with the increase in the use of Multimedia databases, which are usually bigger than traditional databases. In [34] a semantic search engine for multimedia databases namely CROEQS is presented that works as both ontology-based query translator and text based search engine. In relation to the use of ontologies for the provision of intelligent and accurate search engines Kunmei Wen [27] proposed Smartch, which is an ontologybased search engine. In this approach, a ranking method is proposed while searching for concepts, instances and the relationships between them. In Smartch, the end-users' search is performed by keywords. O

- The work carried out in the European TONES project [44] provides relational database access through ontologies. In this approach, data access is enabled by defining links between ontology concepts and relational data. This ontology-todatabase mapping mechanism enables a designer to link a data source to an OWL-Lite ontology. While defining mappings, the designer needs to take into account that an ad-hoc identifier should denote each concept instance so that instance values cannot be confused with data items in the data source. Queries are formulated by consulting ontology-to-database mapping rules, but this rule derivation process is carried out manually by ontology and database experts [44]. Another ontology-based information linking approach with similar techniques, but for query refinement purposes, is presented in [45] and [46]. This approach stores concepts from a data source as part of the ontology and links actual data with ontology concepts. The query answers are improved by using the semantic knowledge expressed in an ontology. Database queries are transformed by using is-a, part-of and sync-of relationships between ontology concepts.

- Query expansion implementations (e.g. [49] and [50]) use thesaurus ontology navigation for query expansion. These approaches use the WordNet ontology (http://wordnet.priceton.edu) for query expansion and adapt basic keyword search mechanisms using keywords, which are identified in the ontology for a matching concept. Another approach based on this thesaurus ontology navigation approach is the Knowledge Sifter [29]. Knowledge Sifter is a scaleable agent-based system that supports access to heterogeneous information sources and relies on the agents technology for query refinement. In this approach, a user query formulation agent supports user query specification to access multiple ontologies using an integrated conceptual model expressed in the OWL. This user query formulation agent also consults the ontology agent to refine or to generalise a query based on the semantics provided by the ontology.

Sampath Kumar, V., Khamis, A., Fiorini, S., Carbonera, J., Olivares Alarcos, A., Habib, M., Olszewska, J. (2019) Ontologies for Industry 4.0. The Knowledge Engineering Review, 34. DOI: https://doi.org/10.1017/S026988919000109.

- The current fourth industrial revolution, or ‘Industry 4.0’ (I4.0), is driven by digital data, connectivity, and cyber systems, and it has the potential to create impressive/new business opportunities. With the arrival of I4.0, the scenario of various intelligent systems interacting reliably and securely with each other becomes a reality which technical systems need to address. One major aspect of I4.0 is to adopt a coherent approach for the semantic communication in between multiple intelligent systems, which include human and artificial (software or hardware) agents. For this purpose, ontologies can provide the solution by formalizing the smart manufacturing knowledge in an interoperable way.

- I4.0 or smart factory (Kannengiesser & Muller, 2013) is based on new and radically changed processes in manufacturing industry. It represents a number of contemporary automation, data exchange, and manufacturing technologies (Hermann et al., 2016), such as virtual enterprise (Smirnov et al., 2010), cloud manufacturing (Xie et al., 2017), Internet of Things (IoT), also named by Cisco as Internet of Everything (Zheng et al., 2014), and its emerging concepts Industrial Internet of Things (IIoT) (Civerchia et al., 2017) or Industrial Internet as used in the US by General Electric (GE) to represent the realization of IoT for industrial applications. In particular, data are gathered from suppliers, customers, and the plant/factory itself and evaluated before being linked up with real production. The latter is increasingly using new technologies such as data analytics, smart sensors, cloud computing, and next-generation robots (Haidegger et al., 2019). This results in flexible and adaptive production processes that are fine-tuned, adjusted, or set up differently in real time (Hermann et al., 2014)

- In particular, ontologies are a powerful solution to capture (Liandong & Qifeng, 2009) and to share the common knowledge (Hoppe et al., 2017) among the distributed partners of the I4.0 technology, leading, for example, to Context-as-a-Service platforms (Hassani et al., 2018). Indeed, ontologies aim to make domain knowledge explicit and remove ambiguities, enable machines to reason, and facilitate knowledge sharing between machines and humans (Persson & Wallin, 2017) and in between machines (Olszewska & Allison, 2018). Moreover, ontologies for the I4.0 are required to be business focused, that is promoting cooperation with customers and partners (Persson & Wallin, 2017) and, on the other hand, meet ontological, autonomous robotic requirements (Bayat et al., 2016). Furthermore, ontologies need to analyze and reuse domain knowledge by using present ontologies (Persson & Wallin, 2017).

For this purpose, in the last decade, ontologies have been developed for one specific industrial domain such as aviation (Keller, 2016), aeropsace (Kossmann *et al.*, 2009), construction (Liao *et al.*, 2009), steel production (Dobrev *et al.*, 2008), chemical engineering (Vinoth & Sankar, 2016; Feng *et al.*, 2018), oil industry (Du *et al.*, 2010; Guo & Wu, 2012), energy (Santos *et al.*, 2018), and electronics (Liu *et al.*, 2005a). Other ontologies have been used for one specific manufacturing process such as packaging (Liu *et al.*, 2005b), process engineering (Wiesner *et al.*, 2010), process compliance (Disi & Zualkernan, 2009), risk management (Atkinson *et al.*, 2006), safety management (Hooi *et al.*, 2012), customer feedback analysis (Kim and Lee, 2013; Daly *et al.*, 2015), organizational management (Grangel-Gonzalez *et al.*, 2016; Izhar and Apduhan, 2017), project management (Cheah *et al.*, 2011), product development (Zhang *et al.*, 2017), maintenance (Haupert *et al.*, 2014), resource reconfiguration (Wan *et al.*, 2018b), and production scheduling (Kourtis *et al.*, 2019). Ontologies have also been focused on one service, for example, ticketing (Vukmirovic *et al.*, 2006), or on one manufacturing concept, for example, information flow (Bildstein and Feng, 2018), information security (Mozzaquatro *et al.*, 2016), and data integration (Yusupova *et al.*).

- Distributed denial of service (DDoS) attacks are designed to knock a website offline by flooding it with huge amounts of requests until it crashes. [https://www.bbc.com/news/technology-53093611]