Compare proposed model CPDC Architecture with Docker Architecture in enterprise integration systems

Masoud Rafighi† and Yaghoub Farjami ††,

University of Qom, Department of Computer Engineering and Information Technology, Iran

Summary
In this paper, the architectures of distributed systems are investigated. Three of the most important and most efficient architectures are compared and their problems will be expressed. Each of these architectures is unable and ineffective to answer the considered deficiencies for distributed systems. It is necessary to have architecture that responses deficiencies and problems.

Key words:
Docker, CPDC Architecture, Software architecture, Distributed systems.

1. Introduction

Customers' requirements control the creation and deployment of software. Customers demand more and better functionality, they want it tailored to their needs, and they want it "yesterday." Very often, large shops prefer to develop their own in-house add-ons, or tweak and replace existing functions. Nobody wants to reinvent the wheel, but rather to integrate and build on existing work, by writing only the specialized code that differentiates them from their competitors. Newer enterprise-class application suites consist of smaller stand-alone products that must be integrated to produce the expected higher-level functions and, at the same time, offer a consistent user experience. The ability to respond quickly to rapid changes in requirements, upgradeability, and support for integrating other vendors' components at any time all create an additional push for flexible and extensible applications[9].
Down in the trenches, developers must deal with complex infrastructures, tools and code. The last thing they need is to apply more duct tape to an already complex code base, so that marketing can sell the product with a straight face.
Software Architecture [26,2,25] describes the high-level structure of a system in terms of components and component interactions. In design, architecture is widely recognized as the provider of a beneficial separation of concerns between the gross system behavior of interacting components and that of its constituent components. Similarly this separation is also beneficial when considering deployed systems and evolution as it allows us to focus on change at the component level rather than on some finer grain.
For instance, previous work described some of the issues involved in specifying a limited form of dynamic software structure for distributed systems in which the set of components and their interaction change as execution progresses and the system evolves [23]. A change in the software architecture could occur either as the result of some computation performed by the system or as a result of some external management action such as insertion of a new component and change of those connections within the system to accommodate the new component. Management actions are performed by a configuration manager [24] which maintains an overall view of the structure of a system in terms of components and their interconnections and performs changes in the context of that view. In essence, the configuration manager is responsible for ensuring that an executing system conforms precisely to its architectural specification. This approach can however be too restrictive for current dynamic, open systems.

2. Software architecture

Architecture is the fundamental organization of a system consisting of components that each of which is associated with each other and with the system and the principles governing its design and evolution. Software architecture is in fact the selection of a general structure for implementing a software project based on a set of user requirements and business of software systems in order to be able to implement the intended applications and also to optimize and accelerate the quality of software, its production and maintenance. Nowadays due to the development of distributed systems that are constantly changing, the need for a flexible architecture can be felt more than ever [10,11].

Manuscript received February 5, 2016
Manuscript revised February 20, 2016

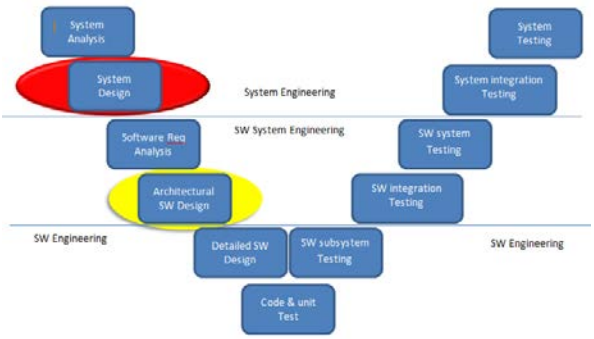

Fig. 1 Architecture: place in system development cycle [10,11].

3. Distributed systems

A distributed system is essentially a computer system where components of the system are held on physically separated, autonomous computers. These machines communicate through the use of a computer network, either a fixed or, in the case of mobile applications, a wireless network. The distributed systems appear to users as a single, integrated computing facility [2,6].

In recent years, distributed systems have become increasingly popular and important in modern computing. They provide opportunities for increasing the reliability, availability and performance of applications. However, perhaps the most important feature of a distributed system is that it allows the integration of existing systems. Companies do not wish to rewrite large numbers of legacy applications and a distributed system allows these applications to be integrated in a relatively straightforward manner [3,1].

A distributed system may comprise components written in a number of different programming languages, running on different operating systems on a variety of computer architectures.

In many cases, a distributed system may be cheaper than a single, centralized system. A large number of small, low-power systems may prove cheaper to purchase than a single mainframe or supercomputer. This is the approach employed in Beowulf clusters, which allow a collection of computers to act as a single large computer [7,8].

There are obviously many significant disadvantages to distributed systems. They are much more complicated to design, build and maintain than an equivalent centralized system. There are a large number of possible failures that could occur in a distributed system, far more than would be found in a centralized system. Because of this, a distributed system will have multiple points of failure, increasing the likelihood of the system not functioning correctly. Communication over a network will always be far slower and less reliable than communication over a local bus, which has a significant effect on the performance of a distributed system [4, 5].

- Distributed systems architectures
  - Client--server architectures
    - Distributed services which are called on by clients. Servers that provide services are treated differently from clients that use services.
  - Distributed object architectures
    - No distinction between clients and servers. Any object on the system may provide and use services from other objects [14,24].

4. Architectures for development of Software distributed

4.1 Docker

Docker is an open platform for developing, shipping, and running applications. Docker is designed to deliver your applications faster. With Docker you can separate your applications from your infrastructure and treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code [16,20].

Docker does this by combining a lightweight container virtualization platform with workflows and tooling that help you manage and deploy your applications. At its core, Docker provides a way to run almost any application securely isolated in a container. The isolation and security allow you to run many containers simultaneously on your host. The lightweight nature of containers, which run without the extra load of a hypervisor, means you can get more out of your hardware. Surrounding the container virtualization are tooling and a platform which can help you in several ways:

A. getting your applications (and supporting components) into Docker containers
B. distributing and shipping those containers to your teams for further development and testing
C. Deploying those applications to your production environment, whether it is in a local data center or the Cloud.

Docker is perfect for helping you with the development lifecycle. Docker allows your developers to develop on local containers that contain your applications and services. It can then integrate into a continuous integration and deployment workflow.For example, your developers write code locally and share their development stack via Docker

with their colleagues. When they are ready, they push their code and the stack they are developing onto a test environment and execute any required tests. From the testing environment, you can then push the Docker images into production and deploy your code [17,21].

4.1.1 Deploying and scaling more easily

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local host, on physical or virtual machines in a data center, or in the Cloud.
Docker's portability and lightweight nature also make dynamically managing workloads easy. You can use Docker to quickly scale up or tear down applications and services. Docker's speed means that scaling can be near real time [18,21].

4.1.2 Achieving higher density and running more workloads

Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines. This is especially useful in high density environments: for example, building your own Cloud or Platform-as-a-Service. But it is also useful for small and medium deployments where you want to get more out of the resources you have.
Docker uses client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running and distributing your Docker containers. Both the Docker client and the daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate via sockets or through a RESTful API [19,20].

Fig. 2 Docker Architecture [16,20].

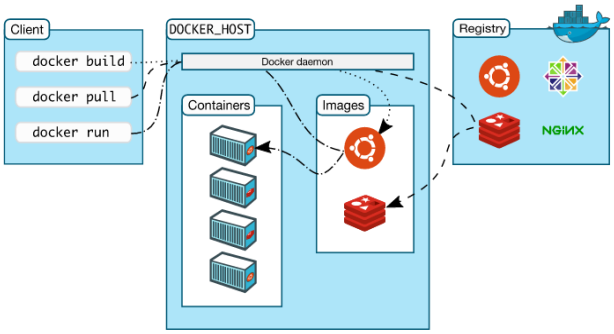

Fig. 3 structure of Docker [16,20].

The Docker daemon

As shown in the diagram above, the Docker daemon runs on a host machine. The user does not directly interact with the daemon, but instead through the Docker client.

The Docker client

The Docker client, in the form of the docker binary, is the primary user interface to Docker. It accepts commands from the user and communicates back and forth with a Docker daemon [20,22].

Inside Docker

To understand Docker's internals, you need to know about three components:

- Docker images

A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created. Docker images are the build component of Docker [20,22].

- Docker registries

Docker registries hold images. These are public or private stores from which you upload or download images. The public Docker registry is provided with the Docker Hub. It serves a huge collection of existing images for your use. These can be images you create yourself or you can use images that others have previously created. Docker registries are the distribution component of Docker [20,22].

- Docker containers

Docker containers are similar to a directory. A Docker container holds everything that is needed for an application to run. Each container is created from a Docker image. Docker containers can be run, started, stopped, moved, and deleted. Each container is an isolated and secure application platform. Docker containers are the run component of Docker [20,22].

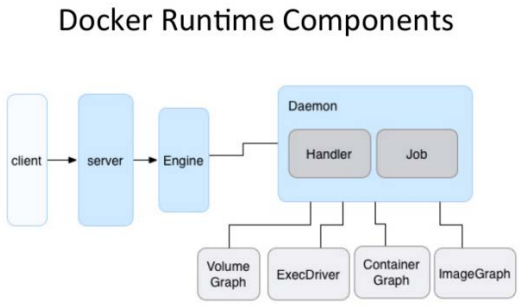

Fig. 4 Docker components structure [16,20].

4.2 The proposed model CPDC Architecture

This architecture made of combining Data- centric architecture, plug-in architecture and component architecture so that in this architecture all components are connected to the data center but the components must appear with two hands (it is getting from plug-in architecture with this innovation that both hands SERVICE INTERFACE and Plug in interface added to every component. It means components have two hands instead of one hand). So in addition to connection they can transfer services and data.By using SOC discuss we concluded that every component must maintain its own data and just Common data such as Authentication and etc. will be kept in Data- Centric. We called the proposed architecture, CPDC Architecture which contains bellow parts:

- **Data center:** Information in the data center, public data, such as user categories, authentication and organizational chart of the organization need to be placed in the center [13,15].
- **Service interface:** An interface to transfer services from one component to another component.
- **Plug in interface:** Certain protocol for connecting components.
- **Service:** Services and operations that are performed on the data in each module.
- **Plug in manager:** management, control and configure of plugin will done.
- **Specific data:** Data that is for a special system and there is no need to exist in other systems.
- **Host component:** The various modules which are available in the organization [10,11,12].
- **ESB:** this part cause strengthened and better performance processes between components.

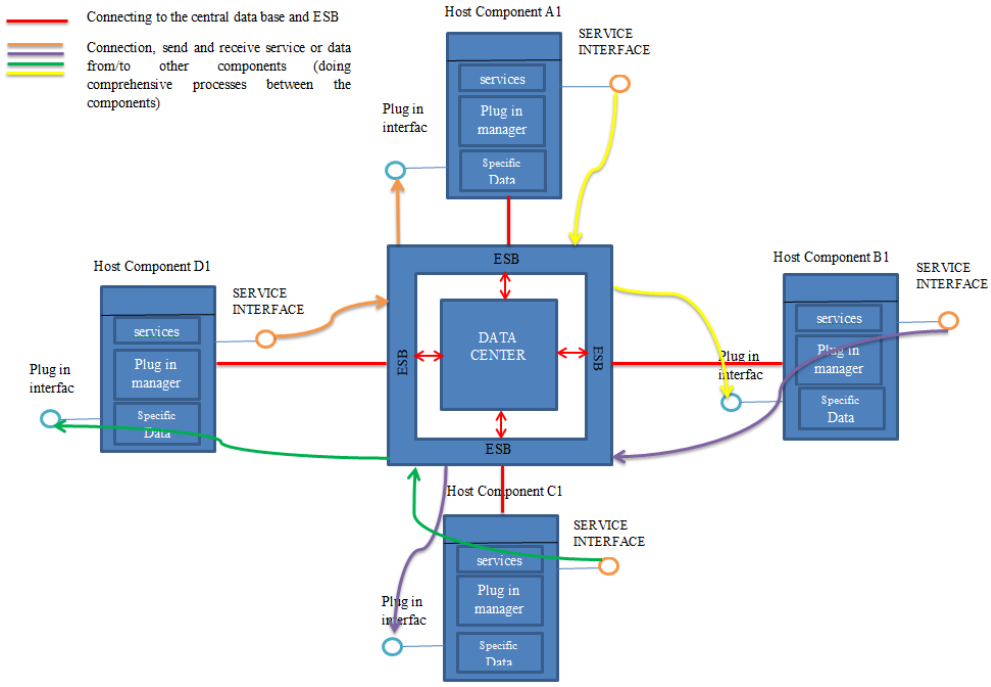

Fig. 5 Proposed model, CPDC Architecture.

5. Measurement and analysis of the architecture criteria

5.1 Layout of components:

Components, as the original block and computational entities participating in the construction of system through internal computation and external communication do their choruses. Every component communicates with environment by one or more port. A user interface can be a common variable; the name of a procedure which calls from other component; it is a set of events that can occur as a component and other mechanisms. Properties of

a component, specifies data for analysis and software implementation.

5.2 Create

Configuration is a connected graph which is sometimes referred to as the topology composed of components and connectors and describes the structure of architecture.

5.3 Connection

When connector makes a connection between two components, component defines an interface. And every component can have several interfaces. An interface is concerned to just one component and every interface of one component can connect to several interfaces in other components. For example in Bus-Oriented architecture the interface of every component is connected to the bus connector and so it will be connected to several interface in other components. Attributes can also be indicated by some of the features, such as communication, buffering capacity and so on.

5.4 development

Development and promotion in computer systems will cause the development and software update. Therefore an important metric that can be considered in the selection of the architecture is extensible metric. The software architecture must be extensible. We evaluate it since this metric has a major role in architecture.

5.5 The main advantage

Each
of software architecture has advantages compared to other architectures. The software architecture eliminates defects in other architectures and complements previous architectures.

5.6 The main problem

Although each of software architectures is trying to be the best and perfect, but, in spite of the development and expansion of information systems, they are still facing problems and in some cases, some complications.
These criteria were chosen only for the problems and shortcomings of Distributed software development architectures and of course there are other factors and criteria that are not effective in this research. To see a full description and explanation of software metrics can be [M. Shaw and D. Garlan, 1996] presented [27].

6. Compare architectures

Table 1. Compare architectures.

Architecture / Criterion	Docker	CPDC
Layout	Docker is an open platform for developing, shipping, and running applications. Docker is designed to deliver your applications faster. With Docker you can separate your applications from your infrastructure and treat your infrastructure like a managed application. Docker helps you ship code faster, test faster, deploy faster, and shorten the cycle between writing code and running code. Docker does this by combining a lightweight container virtualization platform with workflows and tooling that help you manage and deploy your applications.	Information in the data center, public data, such as user categories, authentication and organizational chart of the organization need to be placed in the center. every component must maintain its own data
Creation	It makes an image Of an application with all the configuration settings then you can quickly load and run it anywhere	This architecture is based on component architecture and plugin architecture.
Connection	Docker is not depending to special issue and is usable in all systems. Main part of dacker is: • Docker images. • Docker registries. • Docker containers.	Components have two hands (it is getting from plug-in architecture with this innovation that both hands SERVICE INTERFACE and Plug in interface added to every component). So in addition to connection they can transfer services and data.

Development	It sets a an application include data base, data specification, image, with all the configuration settings on a box that all of the program can work to gather then you can quickly transfer and develop it anywhere	Every system can add easily as a new component if include: Service interface Plug in interface Plug in manager Service Specific data
Elected or a combination of other architectures	It is the infrastructure for using components. It is selected from component architecture and client/server.	This architecture made of combining Data- centric architecture, plug-in architecture and component architecture
The main advantage	The advantages of doker: Software portability, separate the processes, Consumption of resources management, Requires fewer resources, easy use, it is Based on programmer, it solves many of problems of coding and specially dependencies.	Easy extended, in a way that in addition to connection they can transfer services and data and implement comprehensive processes between the components.
The main problem	Program of different boxes cannot be available to each other files and processes between the components don't implement	Security issues and BAM has not been seen in this architecture.

7. Conclusion

By comparing these two methods with parameters (layout, create, connect, development, main advantage, the main problem) we conclude that due to the lack of dependence on the docker architecture, system can be easily extended which it shown on fig.4, but this is not included the extendibility of processes between the components and docker does not have an answer for implementation of processes between the components but in CPDC structure is:

Using with docker

In this architecture using standard protocols soap, an xml-based messaging feature and WSDL document that describes the state of the service, is based on XML, the formal contract between the provider and the service consumer will be increase the integration of system, So that the ESB as a biztalk server and the data centric as active directory operate.

So in this architecture with combining Data- centric architecture, plug-in architecture and component architecture and using SOAP and WSDL standards we can develop system and implement comprehensive processes between the components. Its External view is shown on fig.7.

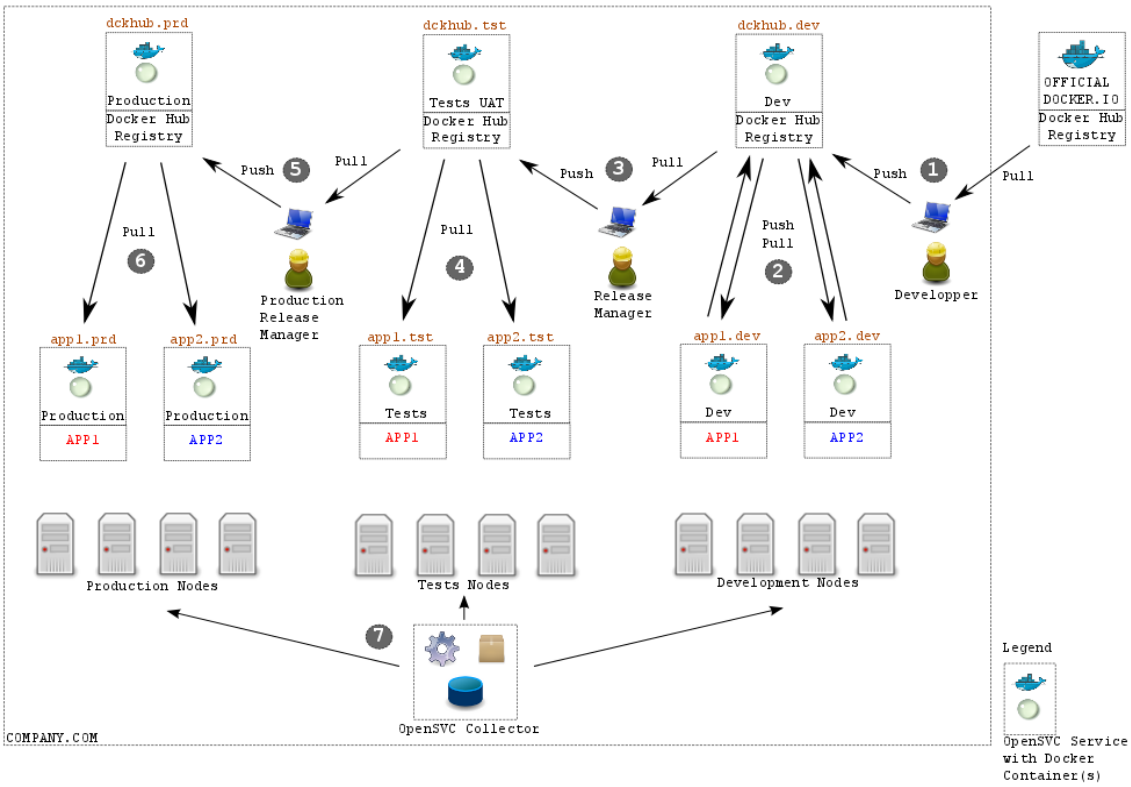

Fig. 6 Relation of component in docker

Using with CPDC Architecture

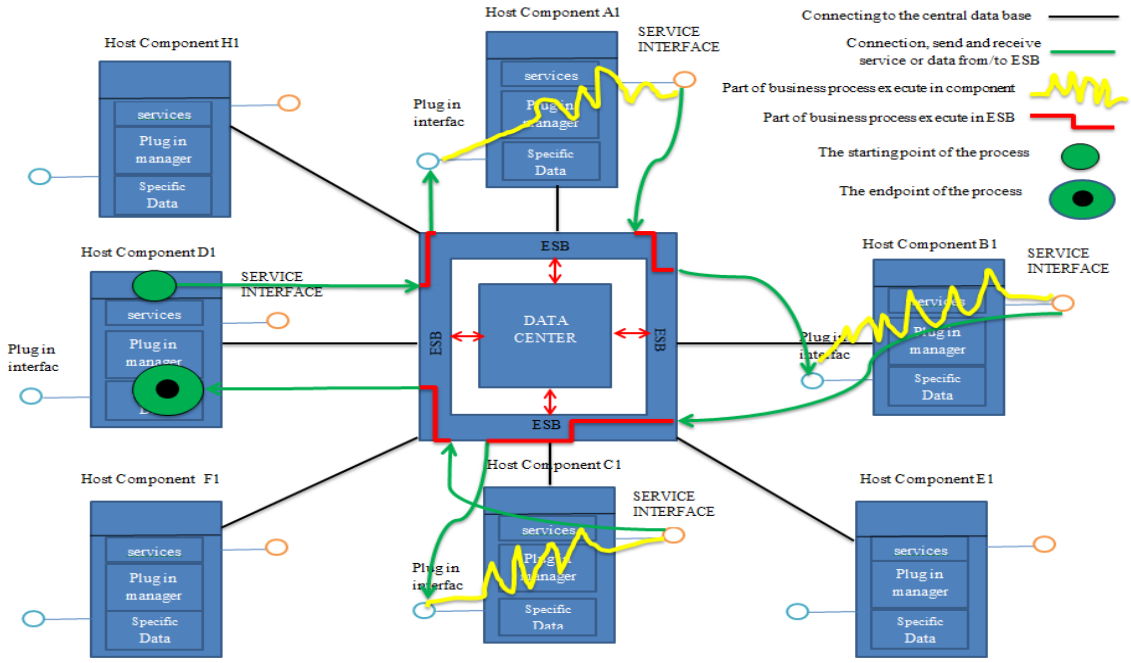

Fig. 7 doing comprehensive processes between the components

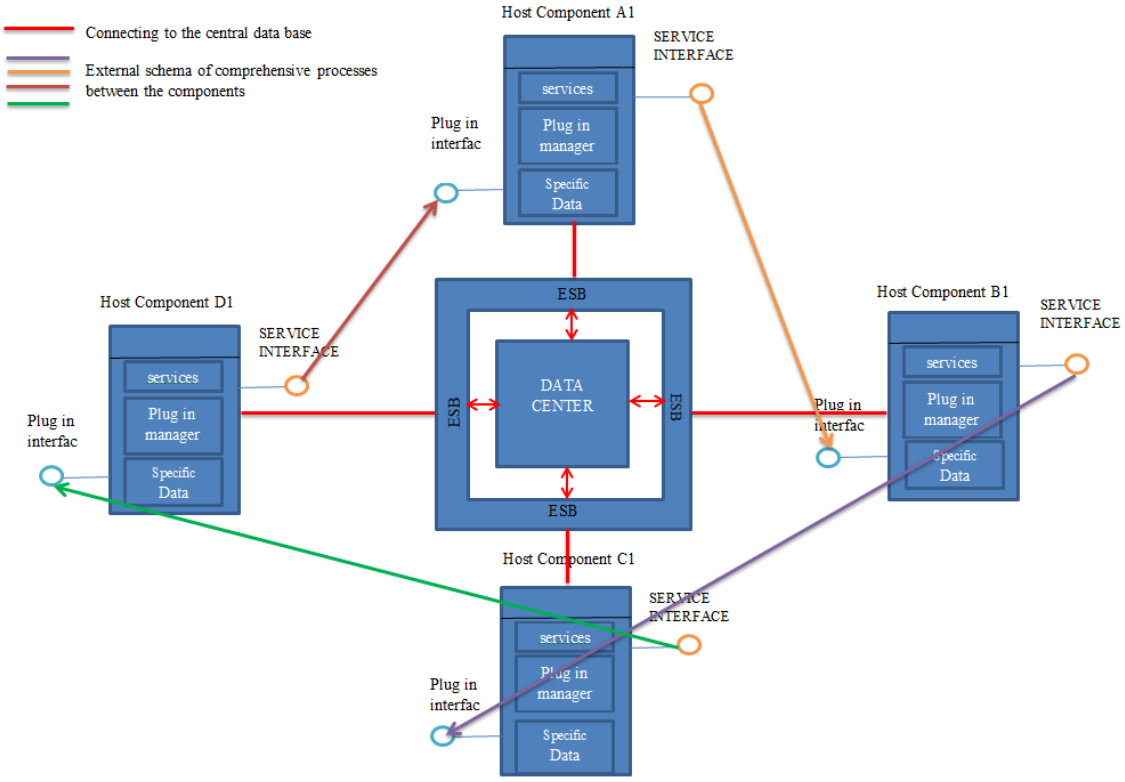

Fig. 8 External view of comprehensive processes between the components

References

[1] Masoud Rafighi, Yaghoub Farjami, Nasser Modiri, Assessing component based ERP architecture for developing organizations, International Journal of Computer Science and Information Security, vol-14-no-1-jan-2016, Pages 72-92.

[2] ERP SYSTEMS: PROBLEMS AND SOLUTION WITH SPECIAL ,REFERENCE TO SMALL & MEDIUM ENTERPRISES, Indu Saini,Dr. Ashu Khanna,Dr. Vivek Kumar, International Journal of Research in IT & Management, IJRIM , Volume 2, Issue 2 (February 2012)

[3] The Future of ERP Systems: look backward before moving forward, Ahmed Elragal , Moutaz Haddara, CENTERIS 2012 – Conference on ENTER prise Information Systems / HCIST 2012 – International Conference on Health and Social Care Information Systems and Technologies,ELSEVIER, Procedia Technology5(2012)21 – 30

[4] SaaS Enterprise Resource Planning Systems: Challenges of their adoption in SMEs, Jacek Lewandowski, Adekemi O. Salako, Alexeis Garcia-Perez, IEEE 10th International Conference on e-Business Engineering, 2013

[5] Custom Development as an Alternative for ERP Adoption by SMEs: An Interpretive Case Study, Placide Poba-Nzaou & Louis Raymond, Information Systems Management, 02 Sep 2013.Published online: 21 Oct 2013

[6] Self-development ERP System Implementation Success Rate Factors Analysis, Liu Chen, Liu Xinliang, IEEE XPLORE Symposium on Robotics and Applications(ISRA) 2012

[7] McIlroy, Malcolm Douglas (January 1969). "Mass produced software components". Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Scientific Affairs Division, NATO. p. 79.

[8] Ralf H. Reussner, Heinz W. Schmidt, Iman H. Poernomo,"Reliability prediction for component-based software architectures", Journal of Systems and Software, Volume 66, Issue 3, 15 June 2003, Pages 241-252

[9] D. Bennouar, T. Khammaci, A. Henni, A new approach for component's port modeling in software architecture, Journal of Systems and Software, Volume 83, Issue 8, August 2010, Pages 1430-1442

[10] Majdi Abdellatief, Abu Bakar Md Sultan, Abdul Azim Abdul Ghani, Marzanah A. Jabar ,A mapping study to investigate component-based software system metrics, Journal of Systems and Software, Volume 86, Issue 3, March 2013, Pages 587-603

[11] Manuel Oriol, Thomas Gamer, Thijmen de Gooijer, Michael Wahler, Ettore,Ferranti, Fault-tolerant fault tolerance for component-based automation systems, in: Proceedings of the 4th International ACM SIGSOFT Symposium on Architecting Critical Systems (ISARCS 2013), Vancouver, Canada, 2013.

[12] William Otte, Aniruddha S. Gokhale, Douglas C. Schmidt, Efficient and deterministic application deployment in component-based enterprise distributed real-time and embedded systems, Inf. Softw. Technol. 55 (2)(2013) 475–488.

[13] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, Ivica Crnkovic´,A Component model for control-intensive distributed embedded systems, in:Michel Chaudron, Clemens Szyperski, Ralf Reussner (Eds.), Component-Based Software Engineering, Lecture Notes in Computer Science, vol. 5282, Springer, Berlin/Heidelberg, 2008, pp. 310–317.

[14] Xuehai Tang, Zhang Zhang, Min Wang, Yifang Wang, Qing qing Feng, Jizhong Han, Performance Evaluation of Light-Weighted Virtualization for PaaS in Clouds, Algorithms and Architectures for Parallel Processing,Volume 8630 of the series Lecture Notes in Computer Science pp 415-428

[15] James Turnbull, The docker book, April 25, 2015

[16] Di Liu, Libin Zhao" The research and implementation of cloud computing platform based on docker,IEEE, Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on,19-21 Dec. 2014, 475 - 478

[17] WHITE PAPER "Modern Application Architecture for the Enterprise" January 28, 2016

[18] https://docs.docker.com/v1.9/engine/introduction/understanding-docker/

[19] http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

[20] Na Luo, Weimin Zhong, Feng Wan, Zhencheng Ye, Feng Qian, An agent-based service-oriented integration architecture for chemical process automation, Elsevier, Chinese Journal of Chemical Engineering 23 (2015) 173–180

[21] J. Magee, J. Kramer, "Dynamic Structure in Software Architectures, 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4)," San Francisco, California, USA, 21, pp. 3-14, October 1996.

[22] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M.Sloman and K. Twidle, "Configuration Management for Distributed Systems," Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (ISINM 95), Santa Barbara.

[23] M. Shaw, D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline," Prentice Hall, 96.

[24] D. E. Perry, A. L. Wolf, "Foundations for the Study of Software Architectures, ACM SIGSOFT Software Engineering Notes," Vol. 17, No. 4, pp. 40-52, 1992.

[25] M. Shaw and D. Garlan, Software architecture: perspectives on an emerging discipline. Prentice Hall, 1996.

Masoud rafighi was born in Tehran, Iran on 1983/08/10. He is PHD student of Qom University. He receives M.Sc degree in computer engineering software from Azad University North Tehran Branch, Tehran, IRAN. He has recently been active in software engineering and has developed and taught various software related courses for the Institute and university for Advanced Technology, the University of Iran. His research interests are in software measurement, software complexity, requirement engineering, maintenance software, software security and formal methods of software development. He has written a book on software complexity engineering and published many papers.

Yaghoub Farjami received his PhD degree in Mathematics (with the highest honor) in 1998 from Sharif University of Technology, Tehran, Iran. He is Assistant Professor of Computer and Information Technology Department at University of Qom. His active fields of research are ERP, BI, and Information Security.