IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications

Soheil Qanbari*, Samim Pezeshki†, Rozita Raisi †, Samira Mahdizadeh*, Rabee Rahimzadeh†, Negar Behinaein†, Fada Mahmoudi†, Shiva Ayoubzadeh †, Parham Fazlali†, Keyvan Roshani†, Azalia Yaghini†, Mozhdeh Amiri†, Ashkan Farivarmoheb†, Arash Zamani†, and Schahram Dustdar*
*Distributed Systems Group, Vienna University of Technology, Vienna, Austria
{qanbari, dustdar}@dsg.tuwien.ac.at
†Baha'i Institute for Higher Education (BIHE)
{firstname.lastname}@bihe.org

***Abstract*—The objective of design patterns is to make design robust and to abstract reusable solutions behind expressive interfaces, independent of a concrete platform. They are abstracted away from the complexity of underlying and enabling technologies. The connected things in IoT tend to be diverse in terms of supported protocols, communication methods and capabilities, computational power and storage. This motivates us to look for more cost-effective and less resource-intensive IoT microservice models. We have identified a wide range of design disciplines involved in creating IoT systems, that act as a seamless interface for collaborating heterogeneous things, and suitable to be implemented on resource-constrained devices. The IoT patterns covered in this paper vary in their granularity and level of abstraction. They are inter-related, well-structured design artifacts, providing efficient and reliable solutions to recurring problems discovered by IoT system architects. The authors offer sound advice for designing, building, and scaling with cross-device interactions inherent in complex IoT ecosystems.**

I. Introduction

Design is the use of scientific principles, technical information and imagination in the definition of a structure, machine or system to perform pre-specified functions with the maximum economy and efficiency[1]. The Internet of Things (IoT) means that hardware and software design practices blend into each other. A well-designed IoT application may be composed of utilized edge devices, fine-grained microservices, cloud gateways to connect edge network to the Internet and mobile or web applications for the user to interact with the underlying things.

There are huge opportunities but considerable challenges [2], [3], [4], [5], [6] in designing IoT applications. These challenges range from the provisioning of ultra-low power operation and system design using modular, composable components to smart automation. Furthermore, the advancement in sensor instrumentation requires an efficient stream data processing. There are also hidden opportunities and challenges in monetizing the edge applications. In response to such challenges, we propose a set of reusable and abstract prescriptive design principles or patterns, which aid system architects in modeling and building *context-tailored* IoT applications. The patterns behold an integrated thinking across many facets of design in an IoT application ecosystem by incorporating the wiring approach of Hanmer [7]. This articulates the benefits of applying patterns by showing how each piece can fit into one integrated solution.

With this motivation in mind, the paper continues in Section II, with a brief review of how IoT design patterns are documented. Next, we introduce the diversity of our proposed patterns and how they are related to the edge applications life-cycle in Section III. With some definitive clues on the pattern language convention, we propose an *edge provisioning* pattern in Section IV, showing how the baseline environment provisioning can be automated. Once the baseline container is provisioned, we demonstrate how code can be automatically pushed to the container via a deployment pipeline. The *code deployment* pattern interacting entities are detailed in Section V. Here, we also define how dynamic configuration effects the quality and performance of the service delivery. Next, in Section VI, the orchestration of IoT services, their dependencies and configurations are focused on and presented as an *edge orchestration* pattern. This leads to rendering the edge application by metering its usage along with its underlying resource usage. Such a metering model is proposed as a *Diameter of Things* pattern and is detailed in Section VII. Finally, Section IX concludes the paper and presents an outlook on future research directions.

II. Pattern Language Conventions

Pattern language is intended to describe the solution in a way that is easy to digest. We are incorporating the cloud computing pattern document format defined by Wellhausen[8] and Meszaros[9] as well as the semantics of its graphical elements to define the structure of our IoT design patterns and their interrelations. All IoT patterns comprise the same document sections with the following semantics.

➾ *Pattern Name*: This name is a handle used to abstract and identify a design challenge.

➾ *Problem*: This is a short summary of the pattern, i.e., the driving question in one or two sentences.

➾ *Context*: This section of the pattern documentation describes the setting in which the problem arises. Assessing the pattern's context influences the solution.

978-1-4673-9948-7/16 $31.00 © 2016 IEEE
DOI 10.1109/IoTDI.2015.18

☞ *Motivation Forces*: This basically provides a use-case scenario, in which the problem is likely to happen and therefore the pattern can be used effectively.

☞ *Solution Details*: This section briefly states how the pattern solves the problem.

☞ *Sketch*: It depicts the functionality of the solution or the resulting architecture after application of the pattern.

We have also ensured that the pattern names are abstract-enough to reflect the pattern's intended design. This conforms to the principle of pattern's name cohesion defined by Hanmer[10].

III. IoT Design Patterns

Designing for an IoT is different. Connected devices may use different types of networks and various connectivity patterns. IoT design patterns vary in their granularity and level of abstraction. To create a valuable, appealing, usable, and coherent edge applications, we have to consider design on many different layers. This is a fine-grained design, as it exposes low-level data from the device itself. While documenting the patterns, we also give an overview of existing implemented frameworks to give a clue and a closer touch on the efficiency of our proposed patterns in production.

Methods for automated provisioning, deployment and configuration management of the behavior of edge applications disclosed herein pertain to governance patterns. Setting up the system and getting the devices connected are hard to simplify. IoT governance patterns deal with governing the edge applications life-cycle from definition through deployment. Such patterns govern all aspects of edge applications including their provisioning and deployment mechanisms by applying runtime reconfiguration and allocation and by scheduling policies to deployed edge services.

Last but not least, we will demonstrate how edge applications are considered value-added and metered services. The proposed metering pattern implements a real-time metering of IoT services for prepaid as well as Pay-per-use economic models.

IV. Edge Provisioning Pattern

♣ *Problem*: How can operation managers and developers ensure all of their edge devices are started with a reliable baseline environment, as needed? How can they provision all the devices automatically all at once?

♣ *Context*: IoT devices are usually scattered geographically, sometimes hard to reach and large in number. Operation managers and developers must be able to reconfigure devices or provision new ones in an efficient way and have pre-configured nodes.

♣ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards spread in a region, each controlled by an IoT city hub[11]. At some point, you need to replace your technology stack entirely and provision a new environment remotely. You may also want to add new devices and provision their runtime environment and applications quickly.

It should be able to provision new devices in a way that can be repeated and produce the same results, so that it can be automated. This also contributes to keeping devices fault tolerant via a rollback to the known-good working state of the environment or applications quickly, if provisioning fails.

♣ *Solution Details*: Container-based virtualization is a good choice for provisioning resources, as they contain not only the code but also all other software dependencies, configurations and the whole runtime environment. By transferring the containerized image to a new machine and running it, we have a pre-configured environment with required applications installed along with any software dependencies.

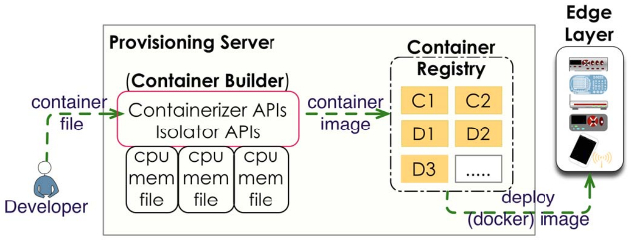

Fig. 1: IoT provisioning pattern sketch.

For instance, provisioning and automated configuration can be done using tools like Puppet[1], Chef[2], or using Docker files, with which the Docker image is built and then transferred to the devices. Docker images utilize a layered and versioned file system, which has two benefits. First, the devices can only pull the layers they need and not the whole image; second, they can rollback to the latest or any working version of the image in case of failure or need. As images are static and read only, this leads to an incorruptible environment for the devices as a backup.

Docker images are built against a Docker file or other aforementioned automated configuration manager, which describes steps needed to build an image as commands to be run inside the container per line of the Docker file. Each step creates a new layer. The image is built; therefore, any resource-intensive step, such as compiling or downloading large files, is done once. Once the image is built, it is pushed to the central Docker registry/hub so that devices then pull the whole image or only the missing layers. These configuration files, like Chef recipes, Puppet manifests or Docker files can be kept as documentation of the steps of provisioning under a version control system such as Git, so that a new commit can trigger the container builder system to build a new Docker image. The edge application together with its environment configuration can be kept in the same Git repository. Therefore, any update to the code or its environment will build a new Docker image and is transferred to the geographically distributed devices, using the same mechanism that is used for deployment. When

[1] https://puppetlabs.com
[2] https://www.chef.io

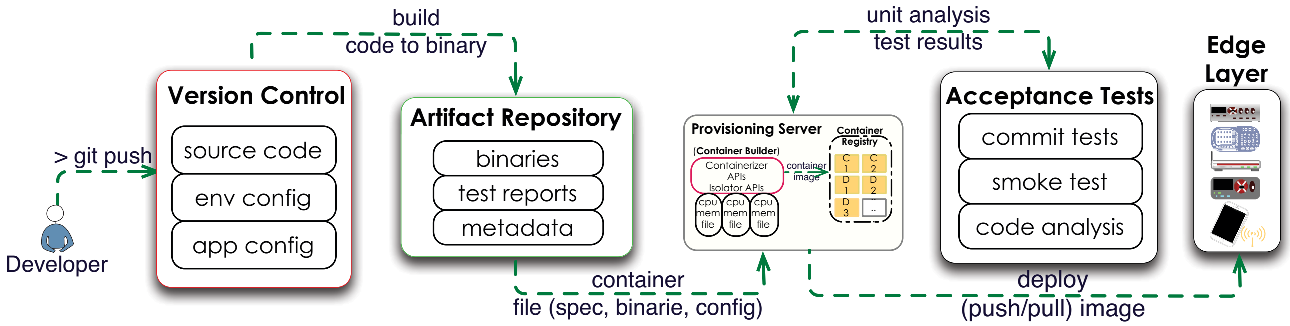

Fig. 2: Edge code deployment pipeline pattern sketch.

the image is delivered to the end device, a new container is created using the image. A sketch of an IoT provisioning pattern is shown in Fig. 1.

V. Edge Code Deployment Pattern

♠ *Problem*: How can developers deploy their code to many IoT devices automatically, quickly and safely, and configure them without being concerned about the long process of build, deployment, test and release?

♠ *Context*: Maintainability is a main factor while deploying a piece of code to some remote IoT devices. As developers enhance and improve the code or fix some critical bugs, they expect to deploy the updated code to their several remote IoT devices quickly. This grants distributing functionality between devices. Also, at some point developers need to re-configure the application's environment.

♠ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards widespread in a region. You need to update the text or graphical features frequently or change the duration of ad display. Maintainability and adaptability are the most important challenges in such designs. You must be able to update the code and deploy it to all your devices at once.

Regarding the poor and flaky Internet connection (e.g., 3G) of a majority of IoT devices, it is best to only deliver the changes and not the application as a whole across the constrained network.

Also, developers should only be concerned with coding, as well as the tools they are familiar with. The tools for deploying the code to devices should be transparent to the developers. This leads to a fully automated deployment. Once automated, the safety of operations increases. The pipeline includes building the application, its deployment and testing and finally releasing and distributing it to edge devices. As for testing, the created image is a production-like environment and is used for testing. Once tests pass, edge devices can pull the image or the corresponding layer. The image is created against all the source code, the container specification and configuration files. Another point is that the developer should be able to rollback its deployment to an earlier version on-the-fly to avoid outage, which is crucial in case of IoT devices.

Moreover, the deployment process should consider and contain any software dependencies or configurations the new code needs. As such, the developer should be able to re-configure the application's environment or the overall technology stack remotely and safely to ensure consistency.

♠ *Solution Details*: As developers are familiar with version control systems, it is best to utilize it for deployments too. Nowadays, Git has become the de facto standard for developers to share their code and maintain versioning. It can be used as the starting point to trigger the build system and then the deployment process.

Git can be utilized by developers to push a specific branch of code to a remote Git repository on the server and is notified of the new version of the software. Then using hooks, it can trigger the build system and start the next step of deploying the code to the devices. The build server builds a new Docker image and pushes the new layers of image to the central Docker registry/hub for the devices to pull. This way the developer only needs to use Git or any other version control system as the only familiar tool for deploying the code into geographically distributed devices.

Devices can periodically ask the central registry/hub for new versions or the server can notify devices about a release of a new image version. Then the devices will pull the new layers of image, create a container from it, and utilize the new code.

In summary, as illustrated in Fig. 2, when the code is modified, it is committed and pushed using Git. Then a new Docker image is built and transferred to devices. Devices use the image to create a container and use the deployed code. The deployment pipeline is started with each commit, and changes in the source code are published to all edge devices.

VI. Edge Orchestration Pattern

♦ *Problem*: How can we orchestrate IoT devices in accordance with their tightly scripted configurations as nodes of a cluster remotely? How can edge cluster nodes discover services?

♦ *Context*: Enabling a large number of devices connected via edge layer means empowering the cluster to manage its nodes to check their health state, their services state to reconfigure them. Moreover, in case of IoT devices to adapt and to calibrate nodes remotely and quickly. Furthermore, we want to be able to manage and run services in the cluster or schedule tasks on certain nodes and enable them to discover the services they need and re-configure themselves accordingly. Edge nodes in an IoT cluster should be able to find themselves and advertise services they provide to each other. Such configuration can be updated over-the-air via WiFi.

♦ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards widespread in a region. You have devices to control each billboard. You want to be able to check their state and health status, manage them, check their services, change their runtime configuration, and execute services in the cluster or on certain devices.

The architecture should avoid having a single point of failure. Each node must be able to know the state of the whole cluster, as well as the state of each service provided by other nodes. In addition, it should manage and adapt itself so that horizontal scaling, replacing nodes and/or adding new nodes or services as ad portlets, in this case ad providers, remains simple.

Such changes in the cluster take effect once nodes are able to advertise their roles and services, as well as to discover each other. Furthermore, we need solutions to allow nodes to reconfigure themselves dynamically according to such changes.

Composite edge applications consist of several inter-related constituents microservices, each in need of its own environment, configuration and even a device. For instance, a home automation system is a composite application. We need a declarative way to describe the whole topology and deploy it considering the orchestration of the components, the services it provides and depends on, without configuring and installing each component separately on every device.

♦ *Solution Details*: Edge infrastructure toolkits treat and provision edge devices with limited compute resources (CPU, memory, and power) as constrained nodes of a cluster. Service discovery mechanisms can be leveraged by nodes to find each other, and the services they provide. Such discovery can also be achieved via device pairing. Once paired, devices trust each other and start sharing data or trigger functionality over a constrained network.

Containers' *compose-oriented*[3] technology enables us to deploy composite applications, since they can orchestrate multi-container IoT microservices. This mechanism expresses the composite application topology along with its specification in a declarative manner. The cluster manager can deploy and orchestrate the composite application for us, according to a service topology specification. This specification describes the composite application, its micro-services and the inter-relationships between them. The cluster manager, which hosts nodes in the cluster, receives the applications' specification and runs services on each node accordingly.

For configuration purposes, distributed `[key,value]` stores can be used. The distributed store, which does not rely on a single node for storing its data prevents a single point of failure. Besides, it enables the cluster to manage itself not only upon node failures, but also upon cluster manager failure. Nodes advertise their services and their state by putting values into the store, others can retrieve the values/updates and establish pairing in one go. This may mean that the values change propagation time could be shortened in TTLs[4] if the nodes get notified of the changes and receive events, instead of polling for updates.

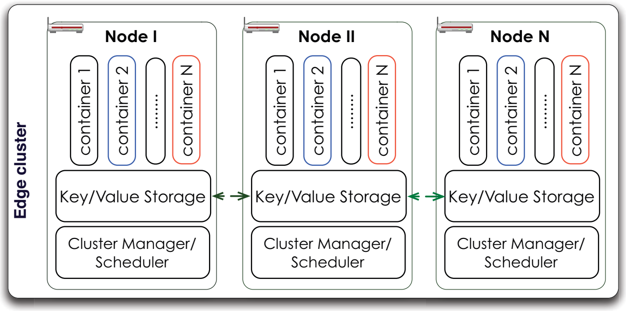

Fig. 3: Edge cluster orchestrator pattern sketch.

From getting the tooling up, we can monitor all of our devices, check the health status of each device, monitor the running services, run new services on all or certain nodes and schedule tasks. When a new value is put into the distributed `[key,value]` store by any node in the cluster, it will be propagated and replicated to all nodes. Then nodes are notified of the new value and discover the services and their state. They then will re-configure themselves. So nodes can advertise the services they provide, find each other or change configurations upon discovery. Note the pattern sketch in Fig. 3 depicting the built-in orchestration model in each node. A downside of this pattern is that the cluster should remain synchronized, meaning nodes should talk to each other periodically. Therefore, network chattering occurs a lot, which is of significance if networking is not economical.

VII. Edge Diameter of Things (DoT) Pattern

♧ *Problem*: How can IoT service provider monitor and meter the actual usage of IoT deployment units in real-time or near-real-time, in order to monetize them? How the IoT composite application resource usage, as well as the service usage can be charged against a specific user balance?

♧ *Context*: From the provider's perspective metering mechanisms can vary based on applied business models. These mechanisms range from different usage patterns such as invocation basis (event-based) and usage over time (time-based), to subscription models such as prepaid and pay-per-use models. This yields to the need for defining some metrics for service and resource usage, which in turn, can be used to measure the consumption of the service and to price it.

♧ *Motivation Forces*: Suppose you have provided an IoT platform, which presumably consists of the hardware (device) and composite IoT services, to your customer, and you would

[3] https://docs.docker.com/compose

[4] https://en.wikipedia.org/wiki/Time_to_live

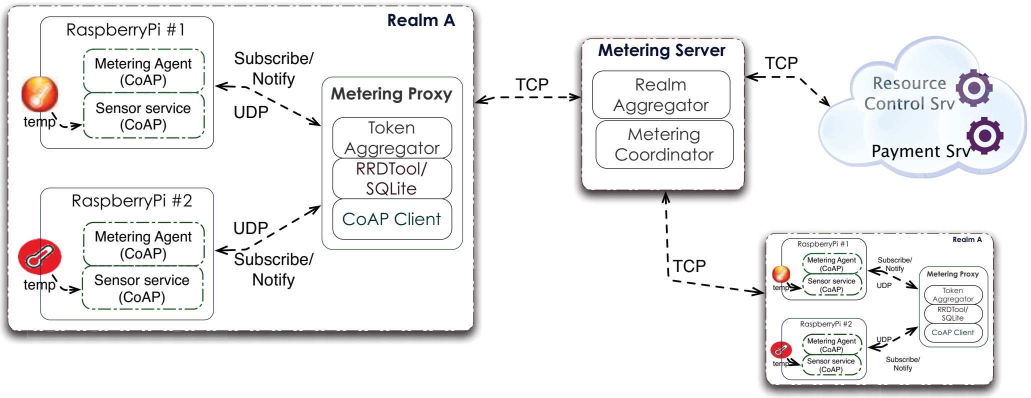

Fig. 4: IoT Diameter of Things (DoT) metering pattern sketch.

like to monetize it based on the real usage of the client, while guaranteeing a fare transaction for both ends. To achieve this goal you need to measure the rate of actual resource and service utilization, as near real-time as possible.

You may provide your service in a prepaid model, where you and the client agree on certain amount of credit to be reserved prior to service delivery. In this subscription model, you should ensure that the actual usage does not exceed the reserved credit on one hand, and on the other hand, the service delivery does not terminate while there are still some credits remaining.

Furthermore, the IoT platform normally resides on the client side and your access to the edge device is limited to an Internet connection with a certain bandwidth. In this regard, you should prevent overwhelming the network by transferring too many monitoring messages back and forth, while still supporting the near-real time monitoring of the usage.

♧ *Solution Details*: A light-weight metering protocol can be utilized to support the telemetry of composite IoT applications deployed on resource constrained devices.

The prerequisite to such a solution is to define a specific agreement called metering plan, offered by the provider and accepted by the client. The subscribed metering plan is an indication of all assumptions that need to be considered for a proper service delivery, continuous and near-real-time usage metering, and the subsequent charging. The plan includes: i) subscription type (i.e. pay-per-use model or prepaid model). ii) list of constituent microservices provided to the client. iii) usage pattern and measurement unit of each service. A time unit is used for duration-based services and number of invocations is used for the services with an event-based usage pattern. iv) the price for each allocated service unit, v) the price for underlying resource usage, vi) the resource Used Unit Update (U3) rate for each service. vii) if prepaid subscription type is selected, then the maximum allocated units to each service is also included in the plan. viii) subscription-fee for prepaid model, and subscription time for a bounded pay-per-use model.

The U3 rate is one of the main factors defining the real-time characteristics of monitoring underlying services. Nevertheless, it exerts an impact on network congestion rate. Consequently, it is crucial for the service provider to assign an appropriate value to it to balance the trade-off between generated network traffic and real time update.

To realize the metering infrastructure, two main components are introduced: (i) the metering server, which is a central component, is responsible for telemetry coordination. (ii) the metering agent, which is a distributed component residing on the edge device and assigned to a microservice, is responsible for collecting usage information and sending it to the metering server. As soon as an IoT application is instantiated for a specific user, the metering server receives a copy of the metering plan. It then parses the plan and calculates the U3 rate for each constituent service in the plan. Together with the calculated U3 value, it then sends the request to start metering each service to its newly assigned metering agent. At each U3 interval, each agent will send the actual service usage, as well as the resource usage to the metering server.

As the usage update summary is received by the metering server, it is used to calculate the charge accordingly. Fig. 4 provides a schematic view on architecting DoTs metering collaborating components. Our DoT protocol[12][5] is currently in development and the ongoing draft is available in IETF as an Internet-Draft submission.

VIII. Related Work

Most of the current proposed patterns are focused from a cloud perspective. Few are designed with an edge-based focus in mind. Hashizume, Yoshioka, and Fernandez[18][19] build a catalog of misuse patterns like Resource Usage Monitoring and Malicious VM Creation. The Amazon Web Services (AWS) cloud patterns[20] documents design principles for applications hosted in the Amazon cloud. Microsoft proposed a set of Azure cloud design patterns[21] claiming to be usable for native cloud applications. The cloud architecture patterns[22] introduces 11 architectural patterns

[5] https://datatracker.ietf.org/doc/draft-tuwien-dsg-diameterofthings

utilizing cloud-platform services. Erl et al.[23][24] present cloud design patterns very briefly. The only article written by Michael Koster[25], entitled “Design Patterns for an Internet of Things”, just defines some basic IoT concepts in a couple of sentences. No pattern format is supported.

In contrast to all the noted related work, we took further steps towards identifying reusable edge applications design constructs. Our presented IoT patterns are still in development, and we are taking the iterative review approach of Harrison[26], “*the language of shepherding*” , to improve the robustness of patterns. The authors (sheep) of the patterns all aspire to receiving feedbacks from edge architects (shepherds) on the proposed constructs effectively.

IX. Conclusion

So far, we have defined four design patterns enabling IoT architects to construct edge applications. As an outlook, our future work includes further extension to the design patterns to support more diverse applications, as well as refining and updating existing ones. We will focus more on patterns to be used for elasticity, resiliency and Software Defined Networking (SDN) patterns for edge computing. This will study the behavior of impacting players, such as competing applications, in making decisions on the allocation of limited resources amongst infrastructure forces of supply and demand.

Just as evolving IoT offerings target a large diversity of systems, we envision that such design patterns may leverage the performance and scalability of edge applications as well as to gaining acceptance as a de facto design standard to give adequate foresight to edge engineers in IoT.

Acknowledgment

The research leading to these results is sponsored by the Doctoral College of Adaptive Distributed Systems[6] at the Vienna University of Technology.

References

[1] Fielden, *G.D.R, Engineering Design*, ser. London: HMSO, 1975.

[2] D. Bandyopadhyay and J. Sen, “Internet of things: Applications and challenges in technology and standardization,” *Wireless Personal Communications*, vol. 58, no. 1, pp. 49–69, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11277-011-0288-5

[3] T. Xu, J. B. Wendt, and M. Potkonjak, “Security of iot systems: Design challenges and opportunities,” in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 417–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=2691365.2691450

[4] D. Fuller, “System design challenges for next generation wireless and embedded systems,” in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE ’14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 1:1–1:1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616608

[5] S. C. Mukhopadhyay, *Internet of Things: Challenges and Opportunities*. Springer Publishing Company, Incorporated, 2014.

[6] H. Lamaazi, N. Benamar, A. Jara, L. Ladid, and D. El Ouadghiri, “Challenges of the internet of things: Ipv6 and network management,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*, July 2014, pp. 328–333.

[7] R. S. Hanmer, “Pattern mining patterns,” in *Proceedings of the 19th Conference on Pattern Languages of Programs*, ser. PLoP ’12. USA: The Hillside Group, 2012, pp. 23:1–23:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821679.2831293

[8] T. Wellhausen and A. Fiesser, “How to write a pattern?: A rough guide for first-time pattern authors,” in *Proceedings of the 16th European Conference on Pattern Languages of Programs*, ser. EuroPLoP ’11. New York, NY, USA: ACM, 2012, pp. 5:1–5:9. [Online]. Available: http://doi.acm.org/10.1145/2396716.2396721

[9] G. Meszaros and J. Doble, “Pattern languages of program design 3,” R. C. Martin, D. Riehle, and F. Buschmann, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ch. A Pattern Language for Pattern Writing, pp. 529–574. [Online]. Available: http://dl.acm.org/citation.cfm?id=273448.273487

[10] B. Di Martino, “Applications portability and services interoperability among multiple clouds,” *Cloud Computing, IEEE*, vol. 1, no. 1, pp. 74–77, May 2014.

[11] R. Lea and M. Blackstock, “City hub: A cloud-based iot platform for smart cities,” in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, Dec 2014, pp. 799–804.

[12] S. Qanbari, S. Mahdizadeh, R. Rahimzadeh, N. Behinaein, and S. Dustdar, “Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications,” http://www.gecon-conference.org/gecon2015/images/papers/qanbari_paper_25.pdf, 2015, [Online; accessed 19-October-2015].

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] R. Hanmer, *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[16] V.-P. Eloranta and M. Leppänen, “Patterns for distributed machine control systems,” in *Proceedings of the 18th European Conference on Pattern Languages of Program*, ser. EuroPLoP ’13. New York, NY, USA: ACM, 2015, pp. 6:1–6:15. [Online]. Available: http://doi.acm.org/10.1145/2739011.2739017

[17] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated, 2014.

[18] K. Hashizume, N. Yoshioka, and E. B. Fernandez, “Misuse patterns for cloud computing,” in *Proceedings of the 2Nd Asian Conference on Pattern Languages of Programs*, ser. AsianPLoP ’11. New York, NY, USA: ACM, 2011, pp. 12:1–12:6. [Online]. Available: http://doi.acm.org/10.1145/2524629.2524644

[19] K. Hashizume, E. B. Fernandez, M. M. Larrondo-Petrie, and E. B. Fernandez, “Cloud service model patterns,” in *Proceedings of the 19th Conference on Pattern Languages of Programs*, ser. PLoP ’12. USA: The Hillside Group, 2012, pp. 10:1–10:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821679.2831280

[20] K. Tamagawa, “AWS cloud design patterns,” http://en.clouddesignpattern.org, 2008, [Online; accessed 19-October-2015].

[21] L. B. M. N. T. S. Alex Homer, John Sharp, “Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications,” https://msdn.microsoft.com/en-us/library/dn568099.aspx, 2014, [Online; accessed 19-October-2015].

[22] B. Wilder, “Cloud Architecture Patterns,” http://oreil.ly/cloud_architecture_patterns, 2012, [Online; accessed 19-October-2015].

[23] “Cloud computing concepts, technology and architecture by thomas erl, zaigham mahmood and ricardo puttini,” *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 4, pp. 37–38, Aug. 2014, reviewer-Gvero, Igor. [Online]. Available: http://doi.acm.org/10.1145/2632434.2632462

[24] A. N. R. P. Thomas Erl, Zaigham Mahmood, “Cloud Computing Concepts, Technology and Architecture,” http://cloudpatterns.org, 2013, [Online; accessed 19-October-2015].

[25] M. Koster, “Design Patterns for an Internet of Things,” http://community.arm.com/groups/internet-of-things/blog/2014/05/27/design-patterns-for-an-internet-of-things, 2014, [Online; accessed 19-October-2015].

[26] N. B. Harrison, “The Language of Shepherding: A Pattern Language for Shepherds and Sheep,” 1999. [Online]. Available: http://www.hillside.net/language-of-shepherding.pdf

[6] http://www.big.tuwien.ac.at/adaptive/program.html