Memory Optimization of a Distributed Middleware for Smart Grid Applications

Stephan Cejka[1], Albin Frischenschlager[1], Mario Faschang[2] and Mark Stefan[2]
[1]*Siemens AG, Corporate Technology, Research in Digitalization and Automation, Siemensstraße 90, 1210 Vienna, Austria*
[2]*AIT Austrian Institute of Technology GmbH, Energy Department, Donau-City-Straße 1, 1220 Vienna, Austria*

Keywords: IoT Application Management, Distributed Smart Grid Applications, Java Virtual Machine, Memory Optimization.

Abstract: In order to exploit the full potential of IoT-enabled power distribution grids, Smart Grid applications are developed. Their operation on resource-constraint automation devices requires for memory optimized operation. In this paper we present field-approved operation and management solutions for Smart Grid applications, based on a distributed middleware. We introduce a new entity to allow for dynamically loading Smart Grid applications within one JVM. Presented experiments demonstrate the reduction of the memory footprint on the physical device.

1 INTRODUCTION

Modern power distribution grids experience an Internet of Things (IoT)-driven evolution. Power grid devices (such as transformers, breakers, and switches) evolve from former passively/manually operated devices into communicating and interacting elements (Yu and Xue, 2016). This evolution allows for a wide range of novel applications with the purposes of efficiency increase and decarbonisation of the power supply system, as well as the creation of novel business cases.

In the next sections, we introduce IoT-enabled power distribution grids, show the architecture of a future smart secondary substation, and present the role of Smart Grid applications on the substation level of smart distribution grids. After that, we show how our work has led us to a memory resource problem at the substation-located automation component. We propose a memory optimization solution and an evaluation in the sections afterwards.

1.1 IoT-enabled Power Distribution Grids and the Role of Smart Grid Applications

A wide range of IoT-enabled power grid devices (e.g., smart meters, smart breakers, electric vehicles, smart storage systems) have already been introduced into the market and being built into the Distribution, DER, and Customer Premises domains (cf., domains of the Smart Grid Architecture Model (CEN-CENELEC-ETSI, 2012)).

A more economically and ecologically effective and sustainable operation can be achieved by using the modern power grid devices to actively operate power distribution grids (Faschang et al., 2017). In order to reach this goal, sophisticated functions and services need to be developed, making use of ICT-connected power grid components and external services (e.g., forecasting, weather information) (Faschang et al., 2017).

The evolution from passively operated to intelligent secondary substations (iSSN) allows for these novel functions (e.g., voltage and (re-)active power control, distributed generation optimization, virtualization, or decentralized market interaction) by having increased computational power and newly attained communication. These functions are realized by distributed software components – so called Smart Grid applications – primarily operated within the iSSN (Figure 1) (Faschang et al., 2017).

All these applications might be interlinked and interacting through a common middleware, which in our case is proprietary software called Gridlink.

Cejka, S., Frischenschlager, A., Faschang, M. and Stefan, M.
Memory Optimization of a Distributed Middleware for Smart Grid Applications.
DOI: 10.5220/0006350403310337
In *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security (IoTBDS 2017)*, pages 331-337
ISBN: 978-989-758-245-5

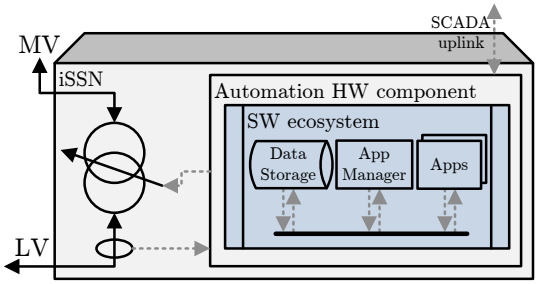

Figure 1: Architectural overview of the iSSN, containing power and communication links, a switchable medium (MV) to low voltage (LV) transformer and an automation hardware (HW) component – typically an industry-grade computer – which operates the smart grid applications (Faschang et al., 2017).

1.2 Memory Usage of Interacting Distributed Applications

In our Smart Grid use case as well as in many other industrial applications, all of the interlinked Java applications currently run on one automation component – an industry grade computer – each of them demanding for a specific amount of computation and memory resources. In the course of distribution grid digitalization, thousands of substations need to be upgraded. Consequently, the cost effectiveness of the automation component is an important aspect. A large number of IoT devices within an application pose for a scalable middleware solution integrating provisioning features.

These considerations can all be mapped to the IoT concepts, where it is also required to achieve distributed functions by use of low priced components. The requirements are almost the same: a distributed application should be able to communicate with its parts, terminologically called modules. The consequences whether two modules are running on the same or on independent devices should be limited. The hardware platform to be used is not restricted, being a significant requirement in the heterogeneous world of IoT devices. It needs however to be able to execute Java SE programs and to have a main memory of at least 512 MB as yielded by field test experiences.

1.3 Outline

In this paper we show that the desired modularity of iSSN applications' modules is somewhat contrary to the limited computation resources that are available and that our previous plans need to be adjusted by a framework enhancement. In Section 2 we summarize our previous work on iSSN applications, the Gridlink and its provisioning features. Section 3 shows that modularity jars with limited resources in the original setup and introduces a framework enhancement to comply. In Section 4, we compare the old with the new solution and show that significant savings in resource consumption have been made. Section 5 finally concludes this paper and shows possible future work.

2 ISSN APPLICATION FRAMEWORK

In previous work, we presented a flexible and modular software ecosystem for iSSNs including a communication infrastructure (Section 2.1) that allows for the operation of distributed applications (Faschang et al., 2016; Faschang et al., 2017; Cejka et al., 2016). Figure 1 shows the software ecosystem of the iSSN including a data storage (Cejka et al., 2015) and the AppManager with a remote uplink for allowing provisioning features (Section 2.2). Other applications include functions for

- acquisition, processing, and analyzing of field component measurement data (e.g., Smart Meter measurements) (Faschang et al., 2016; Cejka et al., 2016),
- linking the iSSN to the energy market (Gawron-Deutsch et al., 2014; Gawron-Deutsch et al., 2015), and
- making decisions, e.g., by a voltage controller application able to switch the transformer's tap-changer based on historical and/or current data (Einfalt et al., 2013; Cejka et al., 2016).

2.1 Gridlink

We introduced the Gridlink – based on vert.x and Hazelcast – as a middleware solution for the iSSN (Cejka et al., 2016; Faschang et al., 2017). It is – unlike many typical IoT middleware solutions (e.g., listed in (Razzaque et al., 2016)) – a distributed middleware with the benefit of not creating a single-point-failure within the system.

Gridlink-based systems are built of several modules (written in Java), each of them communicating with each other by exchanging messages. It uses a distributed event bus based on an asynchronous communication model, improves modules coupling with new functions like a service registry, and enables provisioning features. Modules run within their own Java virtual machine (JVM), termed the node that dynamically form a cluster of known instances during execution using multicast discovery. Modules are able

to join or leave at any time without influencing other modules' execution or communication. A cutback of the overall function of the application on a module's failure is obvious, but can be limited by using redundant modules and reasonable timeouts to react on failed transmissions.

Message Exchange Gridlink message exchange includes the concepts of modules, roles/topics, and services, shown in Figure 2. Modules act as a data source by issuing messages to other Gridlink modules. Messages (i.e., requests or events) are identified by a type (e.g., `createDataPoint`), contain a destination address (e.g., the role `storage`) and may optionally include a payload (e.g., the data point to be created). They are either sent to a designated module role address or published to a topic address reaching all registered modules, usually being marshalled into a JSON representation for transmission. Dedicated proxies are executed after the module called the send or publish command but before the message is really transmitted. This allows for prior executing additional message processing steps, including the modification of a message (e.g., for encryption). The recipient module, registered to the role or topic to which the message was issues to, is the data sink. After the respective proxies are executed on the recipient side, e.g. for decryption, the received message is handed over to the module's service handler and processed, which may include issuing a reply message to the sender. Further process details and an in-depth description of Gridlink proxies are available in (Faschang et al., 2017; Cejka et al., 2016).

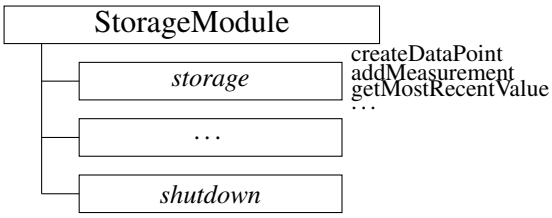

Figure 2: Modules, roles and services (Cejka et al., 2016) A module *StorageModule* is registered for the roles *storage* and *shutdown*. The role *storage* provides several services (e.g., *createDataPoint*).

Gridlink Registry All modules have access to the Gridlink Registry – a distributed list of all modules that are currently attached and active. It includes all roles/topics a module is registered to, a list of requests the according role/topic is able to handle, as well as the proper format of these requests and of possible replies encoded in the JSON schema format. By utilizing the Gridlink Registry, each module knows at any time which modules are currently present and can behave accordingly; by using the defined message schemata even components outside the Gridlink system can establish communication to Gridlink modules.

2.2 Provisioning

In ICT systems at some time it is necessary to install new features, update applications, reconfigure them or remove them at the end of their life cycle. These steps are grouped under the term provisioning, requiring – for cost efficiency – a functionality to initiate these steps directly from the remote operator without requiring staff on site.

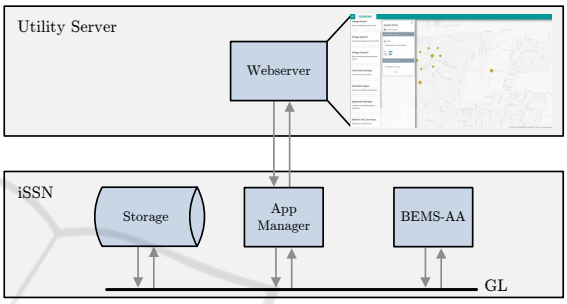

Figure 3: Provisioning tasks are initiated from the remote site, e.g. by an operator's dashboard, communicated to and executed by the local *AppManager* module. Status information is communicated back to the operator.

The *AppManager* module receives commands from remote sites such as SCADA instances at the operator (cf., Figure 3) and manages installation, configuration, updating, and removal of other modules. Hence, these modules are termed managed modules (cf., Figure 4).

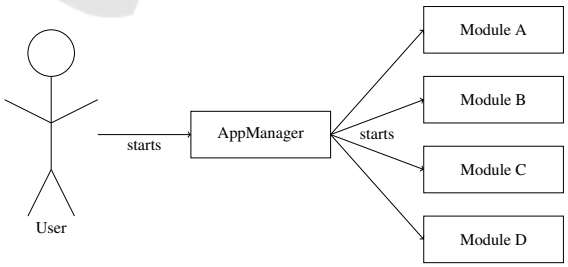

Figure 4: The AppManager and its managed modules (Cejka et al., 2016).

While the modification of a module's configuration shall not require a restart, an update of a module to a newer version shall not lose the module's current state. As the *AppManager* itself is a normal Gridlink module, it is able to communicate with any other module. Since starting and stopping modules is tightly connected with the *AppManager*, selected design

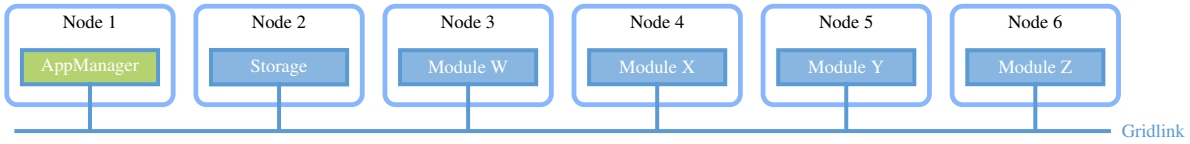

Figure 5: The current solution with one module per JVM/node.

choices of this software will be highlighted, to better understand the later proposed solution.

Start-Up During the *AppManager* start-up, all Gridlink modules in its `managed` directory are started – currently as a new Java process (cf., Figure 5). These modules are then considered to be *managed* by the *AppManager*.

Remote Installation After receiving an installation command from the remote site and the respective download of the new application, the content of the archive is extracted and started.

Configuration Change Configuration changes are done directly on file system level. A new configuration parameter value is written to the configuration file of the corresponding module which is automatically notified.

Remote Uninstallation When a module shall be removed, the *AppManager* sends a shutdown request to the module and waits for the module to disappear from the Gridlink Registry. On success, the module's directory is deleted from the file system. Otherwise, the AppManager can be requested to kill the process over its remote link.

3 OPTIMIZATION

The modularity of the approach requires running a high number of various modules for small independent tasks simultaneously. Previously, one Gridlink node executed exactly one module (cf., Figure 5), i.e. the *AppManager* initializes a new process for each module it is managing, introducing a high overhead in terms of main memory consumption. In consequence, the number of required modules for a typical Smart Grid use case is typically higher than the RAM-constrained devices can host simultaneously, before undesirable effects like trashing occur. The iSSN use case (cf., (Cejka et al., 2016)), currently consisting of seven modules being executed concurrently on the same machine (cf., Figure 6), requires already more than 800 MB of main memory, not available on the target hardware platform. Furthermore, additional modules are currently in development. Thus, a mechanism to execute multiple Gridlink modules in one node and thus in one JVM process had to be developed ("scale up"). The proposed extensions shall however not invalidate any of the Gridlink functions. It shall be possible to run all modules as previously intended, therefore decisions like the start-up of modules, communication etc., should remain.

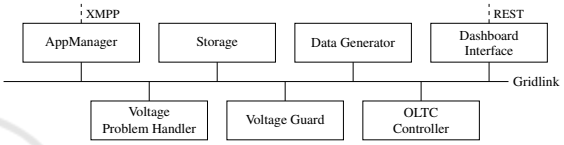

Figure 6: iSSN use case scenario (Cejka et al., 2016).

To solve the described problem of excessive memory consumption, a new Gridlink module – the *NodeManager* – is proposed. It introduces the ability of managing multiple modules on the same node, and thus, in the same JVM instance (Figure 7). To avoid ambiguous use of the term "managed", we will now distinguish between *app-managed* for management by the *AppManager*, and *node-managed* for management by the *NodeManager*, respectively. Therefore, the *NodeManager* is responsible for the module's start-up, provides a command line interface to node-managed modules, and is able to undeploy them. The process of application provisioning is notably influenced by the introduction of a *NodeManager*:

Start-Up The *AppManager* was introduced as an unmanaged module (Cejka et al., 2016). However, besides installing it on its own node (cf., Node 1 in Figure 5), the *AppManager* now can – just like any other module – be started parallel to and by a NodeManager – as a node-managed module (cf., Node 1 in Figure 7). As in the old setup, all Gridlink modules in the *AppManager*'s `managed` folder are started during its start-up by creating a new JVM process, including a *NodeManager* module if it is contained in this folder (cf., *NodeManager* on Node 2 in Figure 7). No special handling is necessary, as the *NodeManager* complies with the module structure conventions. On the *NodeManager*s start-up, all Gridlink modules nested inside its own `managed` subfolder are started within the *NodeManager*s node/process (cf., the remaining

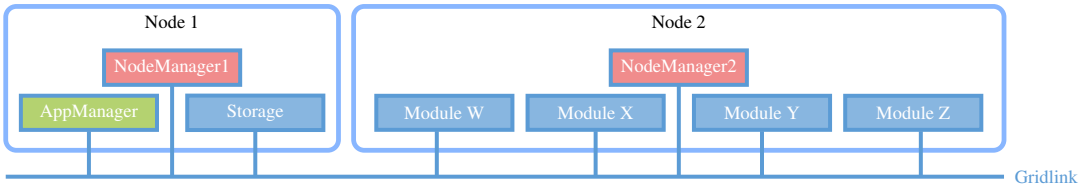

Figure 7: Solution with multiple modules on one JVM/node (”scale up”).
In this example it is decided to execute the AppManager and the Storage module on one node; all other modules together on one other node, respectively.

modules on Node 2 in Figure 7). Special attention has to be paid to libraries of these node-managed modules: Java does not allow for including two classes with the same fully qualified name to the classpath. By using an own class loader for each module, possible problems of influences between libraries are kept to a minimum (cf., isolated bundles in OSGi (Geoffray et al., 2008; Gama and Donsez, 2009)).

Remote Installation The module's configuration needs to include whether it shall be started traditionally as its own process via the *AppManager* or on an existing node next to a *NodeManager*. Based on this decision, the artifact is extracted to the respective directory. In the latter case the request to start the module needs to be communicated to the respective *NodeManager* module.

Configuration Change Configuration changes of a module always reach the *AppManager*, which is responsible for the modification of the JSON configuration file. The location of a module's configuration file depends on whether it is node-managed or not, as it either is located in the *NodeManager*'s or in the *AppManager*'s `managed` directory. The AppManager internally knows all the modules it has started, but not necessarily all modules started by other node managers (e.g. the *Storage* module in Figure 7). However, the AppManager is only responsible for its app-managed modules and thus, all modules of which he is capable of configuration changes are known.

Remote Uninstallation In case of the node-managed module's refusal to shut down, the *AppManager* cannot kill the process as earlier, as this would also shut down the *NodeManager* and every other module on that node. It can however request the managing *NodeManager* to stop the module, shifting the responsibility. As for configuration, for removal of the module's files the *AppManager* has to know the directory location.

Summary In consequence, both *AppManager* and *NodeManager* module are responsible for their ⋆-managed modules. The difference is that app-managed modules are started in their own JVM and thus, have their own process; node-managed modules are started beside the *NodeManager* in the already existing process. Figure 7 shows both options:

1. *NodeManager1* on Node 1 is an unmanaged module, initially started.
2. It has started the *Storage* and the *AppManager*. These two modules are node-managed modules – managed by *NodeManager1*.
3. The *AppManager* on Node 1 has traditionally started *NodeManager2* as its own process on Node 2. Therefore, *NodeManager2* is an app-managed module – managed by the *AppManager* on Node 1.
4. Modules W, X, Y, Z may have either been started (node-managed) by *NodeManager2* or by request of the *AppManager* to *NodeManager2* (in result being app-managed by the *AppManager* and node-managed by *NodeManager2*).

As communication of all modules is done via the Gridlink – and so are the start commands from the *AppManager* to the *NodeManager* – the *AppManager* is also able to request modules to be started on Node 1 by issuing requests to its own *NodeManager1*.

4 EVALUATION RESULTS

To show the impact of the new solution, we conducted experiments using the iSSN use case shown in Figure 6. In the first experiment, each of the seven modules were started in their own JVM recording the aggregated main memory consumption. The second experiment uses the proposed *NodeManager* concept to start all seven modules in one JVM.

Figure 8 compares the results of both experiments. The data show that each JVM requires at least 100 MB main memory. Therefore, the start of the *NodeManager* module in the second experiment shows a

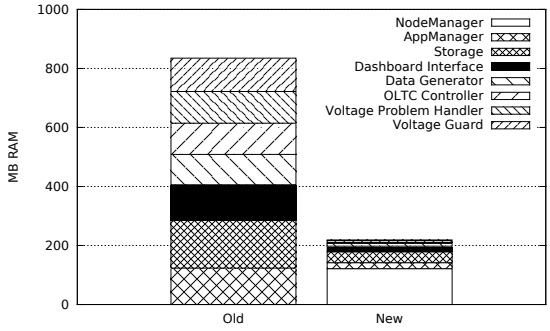

Figure 8: Evaluation results.

comparable memory consumption to each module in the first experiment. More specifically, for n modules with their memory consumption for additional required libraries, its logic, and its data structures m_i and a (for simplification assumed) constant amount of Gridlink dependencies' memory consumption c, the memory consumption for the old solution was:

$$M_{old} = \sum_{i=0}^{n} (c + m_i) = nc + \sum_{i=0}^{n} m_i$$

The memory consumption of the new – *NodeManager*-based – solution requires Gridlink dependencies to be load only once. This results in a significant reduction of the total memory consumption:

$$M_{new} = c + \sum_{i=0}^{n+1} m_i$$

Therefore, the higher the number of modules, the greater the difference of the required memory between old and new Smart Grid application management solutions becomes. However, when only one module should be started on a host, the old system requires less memory. This results from the *NodeManager*'s memory requirements for its own logic and a small Gridlink middleware overhead.

Note, that vert.x – and thus Gridlink – uses an event bus thread shared between all modules running on one node. While this was not significantly relevant for modules if another module got stuck or behaved unexpectedly; the introduction of node management makes this an important – and security relevant – aspect as one hanging module also affects all others on the now shared event bus (Cejka et al., 2017). In consequence, it could be preferred to move modules with higher processor consumption or modules of which a fast reply is necessary, lastly modules counted as fundamental, to their own node without using the *NodeManager*.

5 CONCLUSION AND OUTLOOK

Due to high memory usage of multiple modules running each on its own JVM instance, we introduced a *NodeManager*, which can manage multiple Gridlink modules within one JVM. This reduced memory usage to a huge extent as dependency libraries are not loaded twice. Therefore, no extra JVM-caused overhead is introduced; additionally required main memory just results from a module's implementation.

We raised a potential problem by using modules that block the event bus and thus block every other module on the same node. As modules developed by other parties are possible in our use cases, we need to introduce the ability to certify modules in future work. Joint operation of trusted and untrusted modules on one node may be prohibited.

For various reasons it may be necessary to run modules on different machines ("scale out"). While Gridlink has always been able to communicate beyond the barriers of physical machines, application provisioning is more complicated in this setup. File transfer to and process execution on the other machine needs to be established, e.g. by the use of SFTP/SSH. Being able to deploy applications on other machines, various previously impossible use cases, such as load balancing and the use of fault tolerant and standby modules, are now achievable. Such functionalities are future work, once needed in concrete use cases.

ACKNOWLEDGEMENTS

The presented work is developed in the Smart Grid testbed of the Aspern Smart City Research (ASCR) and conducted

(i) in the "iNIS" project (849902), funded and supported by the Austrian Ministry for Transport, Innovation and Technology (BMVIT) and the Austrian Research Promotion Agency (FFG), and

(ii) in the "SCDA-Smart City Demo Aspern" project (846141), funded and supported by the Austrian Climate and Energy Fund (KLIEN).

REFERENCES

Cejka, S., Frischenschlager, A., Faschang, M., and Stefan, M. (2017). Security concepts in a distributed middleware for smart grid applications. In *Symposium on Innovative Smart Grid Cybersecurity Solutions 2017*, pages 104–108.

Cejka, S., Hanzlik, A., and Plank, A. (2016). A framework for communication and provisioning in an intelligent

secondary substation. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*.

Cejka, S., Mosshammer, R., and Einfalt, A. (2015). Java embedded storage for time series and meta data in Smart Grids. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 434–439.

CEN-CENELEC-ETSI (2012). Smart Grid Reference Architecture. Technical Report, CEN-CENELEC-ETSI Smart Grid Coordination Group.

Einfalt, A., Zeilinger, F., Schwalbe, R., Bletterie, B., and Kadam, S. (2013). Controlling active low voltage distribution grids with minimum efforts on costs and engineering. In *39th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pages 7456–7461.

Faschang, M., Cejka, S., Stefan, M., Frischenschlager, A., Einfalt, A., Diwold, K., Pröstl Andrén, F., Strasser, T., and Kupzog, F. (2017). Provisioning, deployment, and operation of smart grid applications on substation level. *Computer Science - Research and Development*, 32(1):117–130.

Faschang, M., Stefan, M., Kupzog, F., Einfalt, A., and Cejka, S. (2016). 'iSSN Application Frame' – a flexible and performant framework hosting smart grid applications. In *CIRED Workshop 2016*. paper 255.

Gama, K. and Donsez, D. (2009). *Towards Dynamic Component Isolation in a Service Oriented Platform*, pages 104–120. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gawron-Deutsch, T., Cejka, S., Einfalt, A., and Lechner, D. (2015). Proof-of-Concept for market based grid quality assurance. In *23rd International Conference on Electricity Distribution (CIRED)*. paper 1495.

Gawron-Deutsch, T., Kupzog, F., and Einfalt, A. (2014). Integration of energy market and distribution grid operation by means of a flexibility operator. *e & i Elektrotechnik und Informationstechnik*, 131(3):91–98.

Geoffray, N., Thomas, G., Folliot, B., and Clément, C. (2008). Towards a new isolation abstraction for osgi. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 41–45, New York, NY, USA. ACM.

Razzaque, M. A., Milojevic-Jevric, M., Palade, A., and Clarke, S. (2016). Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95.

Yu, X. and Xue, Y. (2016). Smart grids: A cyber-physical systems perspective. *Proceedings of the IEEE*, 104(5):1058–1070.