The current issue and full text archive of this journal is available on Emerald Insight at:
https://www.emerald.com/insight/2633-6596.htm

A framework and method for analysis of feed-forward industrial and manufacturing lines

Received 30 June 2021
Revised 16 August 2021
Accepted 24 August 2021

Cris Koutsougeras
Computer Science, Southeastern Louisiana University, Hammond, Louisiana, USA, and
Mohammad Saadeh and Ahmad Fayed
Industrial and Engineering Technology, Southeastern Louisiana University, Hammond, Louisiana, USA

Abstract

Purpose – This modeling facilitates the determination of control responses (or possibly reconfiguration) upon such events and the identification of which segments of the pipeline can continue to function uninterrupted. Based on this modeling, an algorithm is presented to implement the control responses and to establish this determination. In this work, the authors propose using Message Queuing Telemetry Transport (MQTT), which is an integrated method to perform the system-wide control based on message exchanging among local node controllers (agents) and the global controller (broker).

Design/methodology/approach – Complex manufacturing lines in industrial plants are designed to accomplish an overall task in an incremental mode. This typically consists of a sequence of smaller tasks organized as cascaded processing nodes with local controls, which must be coordinated and aided by a system-wide (global) controller. This work presents a logic modeling technique for such pipelines and a method for using its logic to determine the consequent effects of events where a node halts/fails on the overall operation.

Findings – The method uses a protocol for establishing communication of node events and the algorithm to determine the consequences of node events in order to produce global control directives, which are communicated back to node controllers over MQTT. The algorithm is simulated using a complex manufacturing line with arbitrary events to illustrate the sequence of events and the agents–broker message exchanging.

Originality/value – This approach (MQTT) is a relatively new concept in Cyber-Physical Systems. The proposed example of feed-forward is not new; however, for illustration purposes, it was suggested that a feed-forward be used. Future works will consider practical examples that are at the core of the manufacturing processes.

Keywords Manufacturing lines, Production lines control, Processing nodes, Message queuing telemetry transport, Message exchanging

Paper type Technical paper

1. Introduction

This study concerns the control of complex production lines, also referred to as production pipelines, in which discrete products are incrementally assembled or produced in various stages following specific processing sequences. Researchers have investigated adaptive and smart manufacturing plants to overcome traditional manufacturing challenges, such as diagnostics and predictive maintenance, scheduling, fault tolerance, communication and

© Cris Koutsougeras, Mohammad Saadeh and Ahmad Fayed. Published in *Journal of Intelligent Manufacturing and Special Equipment*. Published by Emerald Publishing Limited. This article is published under the Creative Commons Attribution (CC BY 4.0) licence. Anyone may reproduce, distribute, translate and create derivative works of this article (for both commercial and non-commercial purposes), subject to full attribution to the original publication and authors. The full terms of this licence may be seen at http://creativecommons.org/licences/by/4.0/legalcode.

Journal of Intelligent Manufacturing and Special Equipment
Vol. 2 No. 2, 2021
pp. 75-91
Emerald Publishing Limited
e-ISSN: 2633-660X
p-ISSN: 2633-6596
DOI 10.1108/JIMSE-06-2021-0031

connectivity, as well as many other contemporary issues. Process planning approaches are mostly limited to problems of static nature and are made ahead of their actual use. Thus, their adaptability when unforeseen events take place remain limited and insufficient (Wang, 2013). Distributed process planning (DPP) is a shared cyberwork space where adaptive decision-making that is based on real-time monitoring takes place. Wang *et al.* (2015) proposed a Web-DPP where information is shared with various decision modules to achieve adaptive decision-making. The availability of machining resources and their current status are made available by a monitoring module for dynamic resource scheduling, which in turn helps the Web-DPP for job dispatching to the available machines.

The National Institute of Standards and Technology (NIST) envisioned smart manufacturing as a fully integrated collaborative manufacturing system that responds in real time to meet changing demands and conditions in the factory in the supply networks and in customer needs. Within this definition, the Message Queuing Telemetry Transport (MQTT) protocol would allow for full integration, collaboration among nodes in real time to respond to failures (among other tasks) and to propagate a message to all nodes that sign up to the thread.

There exist several theoretical aspects that deal with fault management and preventive maintenance. These theories typically affect how manufacturing plants and lines are designed. Multiagent systems theory suggests that agents in a manufacturing plant have properties that include autonomy, ability to communicate, reactivity, mobility and decision-making. These agents may also have localized and built-in reasoning mechanism to facilitate intelligent decision-making (Cerrada *et al.*, 2007). Parente *et al.* (2020) outline some challenges that are still facing the manufacturing processes in the context of Industry 4.0. Some of the challenges that are directly related to this work are the need for a decentralized and flexible decision-making and machine proactiveness and self-scheduling. Landers *et al.* (2020) discussed the emerging issues in sensing and monitoring. The work refers to manufacturing processes monitoring in smart manufacturing and the emergence of smart sensors with built-in microprocessors that perform localized decision-making, thus reducing the amount of bandwidth data that need to be processed centrally by the process controller. This gives rise to alternative communication methods that should be versatile and lightweight to allow for swarm communication.

Recently, the concept of Cyber-Physical Systems (CPS) was introduced by Lee (2006). In this model, it is suggested that entities are collaborating and integrating within their surrounding physical world on the cyber space. This necessitates that communication algorithms exist to allow for information exchange between these entities. CPS consists of a hierarchy called the 5C Architecture, introduced by Lee *et al.* (2015b). The proposed MQTT, as suggested in this present work, falls in the second level (Conversion level). MQTT allows for interaction and simple communication exchange among nodes. This allows for performance prediction and for the development of algorithms for prognostics and health management application, which brings self-awareness to machines (Lee *et al.* 2015a). Montostori *et al.* (2016) classify two different types of data that exist within CPS: the configuration data and runtime data. The work cites the importance of real-time communication channel within the CPS to exchange important information about the well-being and availability of these entities. Sensing and monitoring facilitate studying the Overall Equipment Effectiveness (OEE), which combines the operation, maintenance and the management of manufacturing equipment and resources (Dal, 1999). Dal *et al.* (2000) used OEE as an operational measure and as an indicator of process improvement activities within a manufacturing environment.

An early work that depicts a similar message queue system was presented in Syafrudin *et al.* (2018). In their work, they proposed an automotive manufacturing model that depends on IoT to capture vital data of temperature, humidity, accelerometer and gyroscope to

monitor the processes. Their big data processing system consisted of Apache Kafka, which is a message queue system that publishes streams of data using a python-based program that serves as the "producer" for the Kafka server. The "producer" client publishes streams of data to "topics" distributed across one or more cluster nodes/servers called "brokers."

Some of the works reviewed earlier, while related in terms of the application theme, are specific to particular processes and are offered as general reference methodology or case studies. For example, the agents-based design of Cerrada *et al.* (2007) describes such a general reference model for fault management, which provides a general breakdown of the overall management task to subtasks that can be implemented by (software) agents. Some of the other works reviewed above mostly target the prediction of faults rather than the management of faults on occurring.

The particular value of the present work and its contrast with the above is that it provides a Boolean logic-based framework for the analysis of the consequences of a fault(s)—on occurrence—and facilitates the management of the manufacturing system after the fault(s). It provides directly applicable (and easily reusable) methodology and is in line with the CPS concept. In fact, it is also shown here how to implement the methodology as a CPS using a simple message exchange protocol such as the MQTT. The method discussed in this present work is not limited to manufacturing production lines and can be applied to other systems, which accomplish an overall task in incremental steps of sequenced smaller tasks. The benefit of pipelines (production lines) is that a large task is decomposed into more manageable and efficient smaller specialized tasks to improve the overall production throughput. Such pipelines can be simple or very complex; they are usually thought of as a single linear line of sequential tasks, but often they may consist of multiple lines, which feed into other lines as shown in Figure 1.

A stage where pipeline segments merge may indicate different parts arriving at that stage, which then are used for the task of that stage. A problem that may arise in such situations is when a particular stage goes off-line for various reasons, for example, a jam or malfunction at the stage, or for scheduled maintenance; this represents a major issue with pipelined systems. In such a case, the obvious reaction would be to shut down the entire pipeline and effectively freeze it until the malfunctioning stage is serviced. But this may not always be possible or even necessary depending on the structure/design of the system and/or the specific tasks that are performed in the rest of the stages. For example, if one stage is producing a plastic part out of a thermal injection mold, it cannot be stopped while the mold is filled; it has to complete its cycle and eject. However, pipelines are usually designed with some fault tolerance in mind, which may allow partial shutdown of pipeline segments while other segments may still operate, such as in the case of maintenance of urgent shutdown of some stages. In advanced pipeline designs, they may also be dynamically reconfigurable to certain extents. So, the question becomes: given a complex pipeline structure, upon a shutdown of a particular stage, whether or not the whole pipeline needs to shut down or just some segments of it and which ones? Making this determination is obviously important during normal operations, but it is

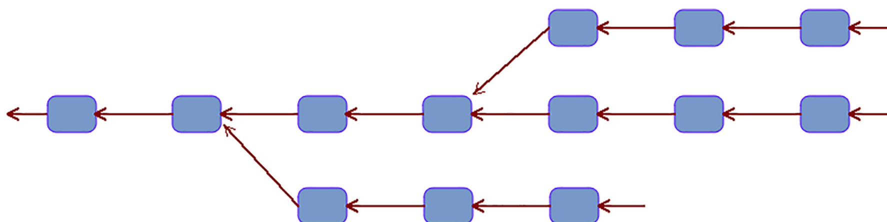

Figure 1.
A pipeline of multiple interacting linear segments

also important to make such determinations on hypothetical events during the pipeline design phase, thereby guiding the design.

Another matter is how exactly the pipeline stages are controlled. It is assumed here that each stage has a dedicated local controller, which controls that stage and that these controllers can communicate among each other. With such an arrangement, it is desirable to have a systematic method for managing the coordination of the controllers at the various stages. If IoT (*Internet of Things*) devices are used to implement the local controllers, then the coordination should not put a lot of extra demand on their computational power; it should be lightweight, yet accounting for all the dependencies among stages. In addition, it should be scalable and fairly easy to implement. This study also includes such a method with an MQTT (MQTT, 2019, 2021) communication framework assumed as the underlying IoT communication infrastructure.

There are other industrial methods to control sophisticated industrial plants with multiple stages and dependencies. For instance, both Distributed Control Systems (DCS) and Supervisory Control And Data Acquisition (SCADA) consist of collections of software and hardware components that allow the supervision and control of plants locally and remotely. They gather information from the plant and processes, analyze them, provide information to the operators and perform tasks with the help of input from operators. In the DCS case, decisions can be made automatically based on messages received from the peripheral components. The main drawbacks for using such systems include: the high cost of implementation, the needs for additional software and compatibility issues and the heavy computation. The MQTT protocol was developed as a more effective solution for networked real-time pipeline data than that of SCADA (Eastburn, 2020).

The present work specifically targets the analysis of the consequences of partial halting events (possibly due to faults or maintenance) in production lines in which discrete objects advance from one processing stage to another. This is useful both in the design phase of production lines and for developing strategies to react to node halts during the operating phase (of which the algorithm presented here is an example). Other works in the literature offer various methods for the analysis of production lines, but mostly target the estimation of throughput and do so with simulation-based models of the production flow. In Phadnis (2013), the author targets a factory layout design based on an evaluation using discrete event simulation to emulate existing factory constraints. This study focuses on the production line layout and material follow to optimize the production time. The study deals with inline automated and manual stations without considering process control techniques in case of failure of any station. Throughput and cost are the main targets in Gao *et al.* (2019) where a method is offered for the analysis of production lines throughput and improve the evaluation efficiency of production lines with various topologies. This can be used to aid the conceptual design phase of production lines. A framework based on generalized semi-Markov processes (GSMP) for the analysis of production lines is offered in Glasserman and Yao (1994). The research focuses on the scheme of the GSMP and related structural properties with application of serial production line. In Bierbooms (2012), a method based on mathematical approximations is presented for the analysis of the performance of production lines, both for the case of discrete item products and the case of continuous flow of material. The research introduced the concept of effective process time (EPT) based on many factors including machine breakdown or uptime and a repair or downtime. Targeting production output volume, Starkov *et al.* (2012) present an analytical model with simulation to prove that the surplus-based decentralized production control strategy is an optimal control policy, which ensures that the cumulative output of products follows the cumulative production demand. In summary, the main focus of previous works has been the analysis or estimation of production flow by means of discrete or stochastic methods.

This work targets specifically the problem of node halting or faulting, the determination of operational consequences and the way to react or control the system after such events. It introduces an alternate method of analysis and control that is lightweight and simple to implement and which could be used in tandem with methods such as the ones referenced above. The following sections explain the logic of the protocol and the method it uses to determine the message-exchanging algorithm. The protocol is simulated using a model of a typical manufacturing pipeline to illustrate the algorithm and demonstrate its effectiveness.

2. Methodology

2.1 Communication protocol

The MQTT communication protocol is well suited for facilitating the management and control of multiple IoT devices when the control can be based on simple discrete message exchanges (Hechtman, 2021; Hasan and Mohammad, 2018; Jaloudi, 2019; Mehmood *et al.*, 2019). The MQTT is a lightweight message exchange protocol that can facilitate the networking of simple and lightweight IoT devices. It is a publish/subscribe messaging protocol according to which clients (IoT node devices) connect to the network by registering with a coordinating device referred to as the network "*Broker.*" A node device (client) can subscribe (with the broker) as a listener or publisher to one or more "*topics.*" A topic essentially is a class/category of messages (usually themed) to which various nodes of the network can declare (to the broker by subscribing to the topic) that they wish to be copied on messages posted in that topic, that is, in that class/category of messages. A client node publishes to a topic by sending the data (to be published) to the broker, which then distributes copies of it to all the node clients that are subscribed to that same topic. So, when a node of the network has a message to communicate to other nodes on a particular topic, it simply sends it to the broker, and the broker replicates the message and sends it to all the network nodes that have subscribed as listeners to that topic. This is done by a lightweight protocol that is much simpler than the standard Transmission Control Protocol (TCP) and has much less overhead in the amount of data transmission. It is also very easily scalable since adding a new client node to the network is as simple as subscribing the new node to the pertinent topics as a listener and/or as a publisher.

To use the MQTT protocol for the purposes of managing a production pipeline, one could consider all the (IoT) controllers of all the pipeline stages as nodes of a network, each of which runs the MQTT client. There must also be one node designated as the broker and (of course) at least one topic to which all the client nodes subscribe as listeners and/or as publishers. In this way, information about a jam at one node can be broadcasted to all other nodes, which then should react accordingly. But, what should the reaction be? One alternative is to define the reaction as: shutdown everything, that is, broadcast a message to all nodes to halt. However, in a complex pipeline, especially those designed as manufacturing or production pipelines, there are usually provisions that may allow parts of the pipeline to still operate while other parts are shut down. So the challenge is how to retain the advantages of using the MQTT protocol while identifying the minimal partial shutdown that is necessary to manage a halt at a node.

2.2 Modeling of the pipeline

A model of the pipeline operation is needed for the study of methods to handle node halts as well as for simulating various design considerations or failure scenarios. Such a model has to reflect the way a node's operation depends on other nodes and thus how one node affects others. To develop such a model, we start at the fundamentals of a pipeline operation:

(1) There are no loops in the pipeline. That is, there is no feedback – in the sense that no item comes back to the same stage (node) for further processing. In this way, the pipeline is assumed to be a feed-forward-only structure as shown in Figure 2.

(2) Each node receives one or more inputs (feeds) from other nodes. In the case of multiple inputs to a node, there are few considerations:

- All of the inputs are needed for a processing cycle at that node. As far as dependencies are concerned, the function of this node depends on receiving feeds from all the input lines. In modeling the dependency of this node from its input feeding nodes, this would be represented as a logical AND dependency.
- One input is an alternate to another input. This would represent two alternate sources of the same part needed for the local operation at that node; for example, the same type of part is supplied by two different pipelines arriving at this node and supplying it with the same part but from different sources (this type is discussed later in section 3.1). In modeling the dependency of this node from its input feeding nodes, this would be represented as a logical OR dependency.
- A combination of the above arrangements may exist in a set of inputs to a node. For example, four inputs feed into a node of which two (I_{A1}, I_{A2}) are alternate suppliers of the same part A, the other two (I_{B1}, I_{B2}) are alternate suppliers of the same part B and both parts A and B are necessary for the operation of this node. In modeling the dependency of this node from its input feeding nodes, this would be represented as a logical AND-OR dependency: (I_{A1} OR I_{A2}) AND (I_{B1} OR I_{B2}). According to Boolean logic, this would be equivalent to (I_{A1} AND I_{B1}) OR (I_{A1} AND I_{B2}) OR (I_{A2} AND I_{B1}) OR (I_{A2} AND I_{B2}). For simplicity of notation, we will denote the operator AND with a "$*$" and the operator OR with a "+" as is usually the case in Boolean logic notation.

(3) A node's output is typically fed to only one other node (for further processing), but it is also possible that the output items are routed to more than one other nodes in a round-robin fashion. This might be the case when a stage produces more throughput than

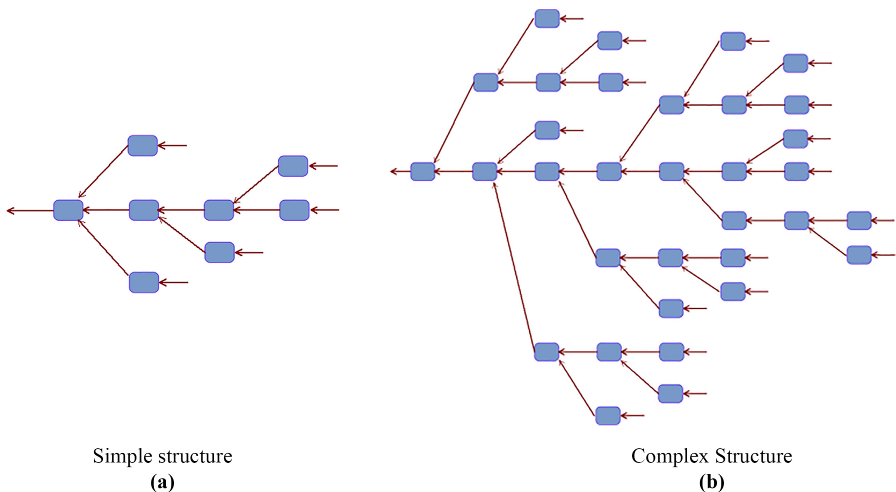

Figure 2.
Examples of feed-forward structures; no feedback

the processing rate of the next stage in the pipeline or when the next stage in the pipeline is shutdown, thus throughput can be routed to a temporary holding location. The concept is illustrated in Figure 3 for node A with two output (receiving) destination nodes B and C. In Figure 3 the output of node A is shown directed by a direction mechanism (selector switch), which alternates the output between the destination nodes B and C; the figure only illustrates the concept of multiple alternate destinations of an output by simple modeling and is not otherwise tied to the specific hardware implementation for the output distribution. The determination of where to direct the output of node A is assumed to be handled by the local controller of node A. A directing guide that rotates to one of two positions is shown for illustration of the concept.

3. Implementation of the algorithm

With the above premise, the dependencies of the model system of Figure 4 can be described as follows:

For normal operation, node *H* needs an input from node *F* and either an input from node *E* or one from node *G*. Node *F* needs either an input from *C* or one from *D*. Node *C* needs both inputs from *A* and *B*. With this graph, it is evident that a halt at node *D* does not need to shut down the rest of the system and the system can still operate with the node *D* offline. Similarly, one of either node *E* or *G* (but not both) would allow the system to operate. A halt at either node *A* or *B* would cause a halt of node *C*, but the rest of the system can still operate. A halt of node *F* would cause the whole system to halt. A halt at node *C* would need to force a halt at both nodes *A* and *B*, but the rest of the system can still operate.

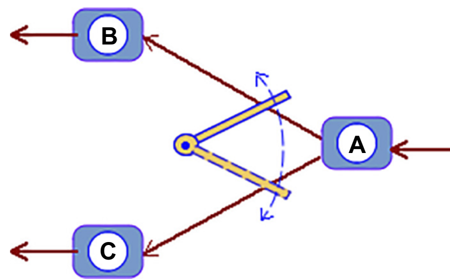

Figure 3.
Multiple output destinations

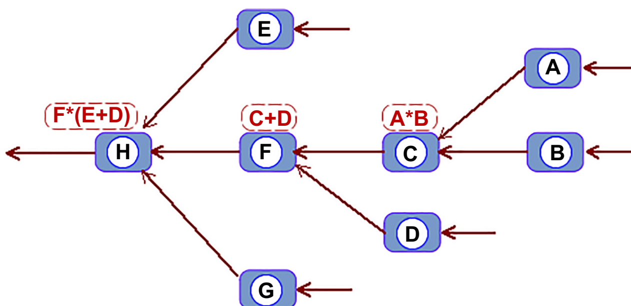

Figure 4.
Model system

So, it is evident that a modeling of the system in this way can reveal what controls are needed for the system and its various nodes in the event of a particular node halting. It is only a matter of developing algorithms for traversing the logic of the model in order to identify what control actions are needed. In section 3.5, we explain how this model can be represented in a JSON (JavaScript Object Notation) computing structure, which then can be used by the system controller running the control algorithm(s) in any language.

3.1 Use of the model

In this section, we introduce the observations that can be derived on the basis of this modeling. Suppose that in the system segment depicted in Figure 5, node *A* halts. What happens with node *C* to which node *A* feeds (provides input)?

Considering the type of dependency of node *C* from node *A*, we observe the following:

(1) If the dependency of node *C* from *A* and *B* is an OR dependency, then node *C* will be able to continue operating even without *A* producing an output. We conclude that *C* will be unaffected for an OR dependency from *A*.

(2) If the dependency of node *C* from *A* and *B* is an AND dependency, then:

- Node *C* will not be able to continue operating without *A* producing an output. We conclude that *C* will also need to halt for an AND dependency from *A*.
- Because node *C* will need to halt, node *B* will also not be able to push its product out and will need to halt.

The above (re)actions will have to be applied recursively to the entire structure because as it was evident above, node *B* (although seemingly independent from *A*) may need to halt if *A* halts.

Now consider a more involved case shown in Figure 6 in which node *A* halts. The substructure of the nodes *A*,*B*,*C* is assumed similar to the structure of nodes *A*,*B*,*C* of Figure 5

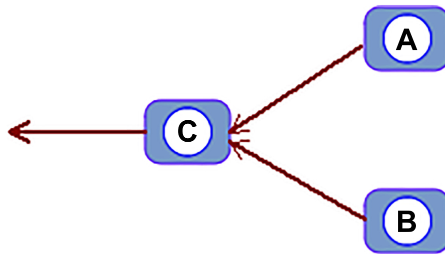

Figure 5.
Basic analysis

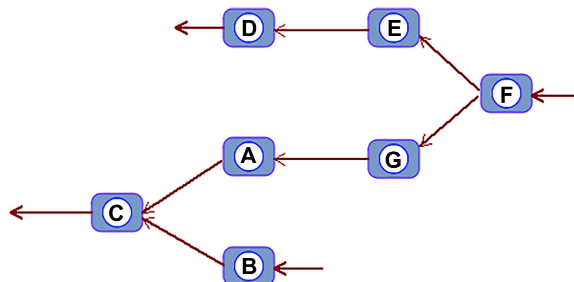

Figure 6.
Basic analysis continued

but in this example there is a node F that alternates its output to two other nodes (E and G). Although node F can feed both E and G, its output at any given time can only go to one receiving node, either E or G. For this reason, there is no need to contemplate any AND/OR designation for such output. The only assumption for such output is that as long as at least one of the nodes E and G is functioning, F can function too and will only need to halt if both E and G halt.

So, considering that in the structure of Figure 6 node A halts, the outcomes for nodes B and C are the same as discussed in the previous example of Figure 5. However, in this example, we are concerned with the pipeline part that is upstream from the halting node A:

(1) Node G will also have to halt if it cannot push its output out to the (only) receiving node A.

(2) With node G halting, F does not need to halt if it can still push its output to the (alternate) receiving node E.

3.2 Simulating the algorithm

The following model (Figure 7), which depicts a complex manufacturing pipeline, will be used to illustrate the effects of node halting according to the analysis introduced in the previous section. This also illustrates that this method of analysis may be used to identify weaknesses of a pipeline design and thus help with the design of more fault-tolerant pipelines. In the following two examples the analysis is informal and qualitative in order to explain the logic, but these help the understanding of a specific algorithm that follows next.

The following analysis scenarios refer to the example structure of Figure 7.

3.2.1 Example analysis 1: Node 9 halts. The feeds from nodes 1 and 2 will be blocked, and since nodes 1 and 2 only feed node 9 and do not have alternative routes, they will both need to

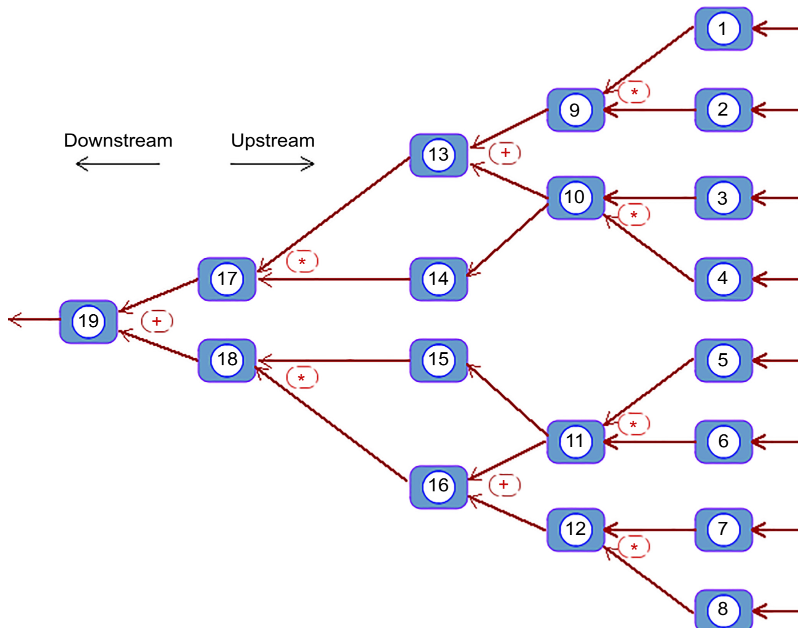

Figure 7.
Example of analysis

be halted. With node 9 halted, we need to examine node 13, whose input dependency is an OR (designated as "+") so it can still function with the feed from node 10. Conclusion: with node 9 halting, nodes 1 and 2 need to also halt but the rest of the system can function.

3.2.2 Example analysis 2: Node 15 halts. On the downstream side, node 11 seems to have an alternate route to node 16, so at this time there is no obvious reason for it to halt. Also, there is no need to continue the upstream trace (nodes 5,6). On the downstream side, node 18 will have to halt since it needs both its inputs (input dependency is an AND -designated as "*"). Since node 18 has to halt, we need to recursively apply the analysis for this new halt event (of node 18). On the downstream side, the affected node is 19, which has an OR input dependency and so it can still function with input from node 17 alone. On the upstream side of node 18, node 16 will have to halt since its output, fed only to node 18, will be blocked. With the new halt event of node 16, we apply the analysis again on the upstream and we see that node 12 will need to halt. Examining node 11, it is now evident that node 11 will have to halt because both its (alternative) output routes will be blocked. The analysis applied again for the new halting nodes 11 and 12 similarly reveals that nodes 5, 6, 7 and 8 will also have to halt. The rest of the system consisting of node 19 and 17 and the subsystem upstream from node 17 can still function. The previous analysis scenarios indicate how the earlier formulations for the nodes' input dependencies can be used to determine the effect of a halting event. Yet, the actual goal is to learn how this can be done algorithmically and thus in an automated way.

3.3 The algorithm for the analysis

The examples given in the previous section lead to an algorithm for checking the effects of one or more nodes halting. In particular, the analysis of example 2 above points to a method, which in computer programming is known as Breadth-First-Search. According to this method, to determine which objects belong to a certain category, one starts with a set of initial objects that are known to be of that category. Then, this set repetitively expands by including new objects, which are determined to also belong to this category based on their relation to objects, which are already labeled as being part of the category. Essentially, the set (representing the category) starts with the known objects and then expands like a bubble to include new objects based on their relation to the already labeled ones. The algorithm terminates when no new inclusions can be made.

Thus, in the case at hand, the way this algorithm works is as follows. The initial set of halting nodes consists of the one that is known to have halted. If more than one node halts simultaneously (e.g. multiple jams happening at the same time), then the initial set consists of all the known halted nodes. In subsequent iterations, the rest of the nodes are examined to determine which others need to halt based on what is already known to halt (i.e. which nodes are already in the category of halting nodes). These iterations continue until no more new nodes are added to the halting category.

In each iteration, to determine any possible new nodes that need to halt, we apply the concepts of the formulation regarding dependencies. This is done by scanning the nodes that have not yet been labeled as halting. For each such node, first we test if all of its output destination nodes are in the halting set. If all of its output streams are going to be blocked, then this node must be added to the halting set. Note that this might subsequently require, in a similar fashion, other nodes to be labeled as halting. Second, for the same node we test if it can function based on which other nodes have been labeled as halting. For this second test, if the input dependency of the currently considered node is an AND, then if any of its feeding nodes is in the halting set, then this node must be added to the halting set (i.e. be labeled as halting). If the input dependency is an OR, then if all of its feeding nodes are labeled as halting, then this node too must be labeled as halting. But even in the more general case where the input dependency is a composite AND-OR logic (Boolean) function, the test is simply the outcome of

that logic function (the node's logic dependency formula) where the inputs corresponding to already labeled nodes are set to logic 0 (halting) and those corresponding to unlabeled nodes set to logic 1 (able to operate). This test is valid even for nodes with a single input (the trivial case), in which case the test obviously reduces down to passing the label of its feeding node (accordingly 0 or 1 as above). This algorithm described here in verbose is described in pseudo-code in section 3.4.

It should be noted here that the algorithm aims to determine what parts of the pipeline are *feasible* to operate after one or more nodes halt. The overall optimization of the pipeline's throughput is not part of the present study, although it can be addressed in future studies.

3.4 The algorithm to check for halting nodes

Table 1 shows one algorithm to determine which nodes of a pipeline will need to halt as a consequence of one (or more) node halting.

For purposes of illustration, the following analysis demonstrates how this algorithm would run on the previous examples 1 (section 3.2.1) and 2 (section 3.2.2):

Case of example 1 where node 9 halts:
The array H is initialized with a 0 for node 9 and 1's for the rest of the 18 nodes ;
In the first iteration of the while loop of line 5:
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
  For node 1, the code at line 11 will set H[1] to 0. Similarly for node 2, H[2] will be set to 0;
  The flag C will be set to 1 (so that the while loop of line 5 will continue) ;
  For node 13, the code at line 10 will set H[13] to (H[9] OR H[10])=(0 OR 1)=1 ;
The first iteration of the while loop of line 5 ends and a new one starts because C is '1':
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
  In this iteration there will be no changes in any H[] values, so C will remain cleared ;
The second iteration of the while loop of line 5 ends and a new one does not start because C is '0' ;
Program ends with H[] values of 0 indicating the halting nodes.

1.	*// Each node is assumed to be represented by a record indicating:* *// List of input nodes; Input dependency as a Boolean function; List of output (receiving) nodes;* *// List of outputs may be constructed at initialization as it is implicitly reflected in the inputs of the receiving nodes* *// Also create an array H[] of true/false (0/1) labels for the various nodes exists to label halting nodes* *// Initialization: Create array H[] with all elements set to '1'.* *// Mark the node (or nodes) halting/failing in the array H[]*
2.	H[Node(s)]=0;
3.	C='1'; *// Boolean flag set to 'true', indicating to enter/continue the following iterations.*
4.	*// Start iterations*
5.	while C is '1' *(i.e. at least one node is newly marked as halting)*, do:
6.	{
7.	C='0'; *// clear flag C*
8.	for each node N where H[N] is '1', do:
9.	{
10.	H[N]=value(Compute the Boolean output of node N's dependency function based on H[i] values);
11.	for each node T that is an output receiving from N do: { if all H[T] values are '0', then set H[N]=0; }
12.	if H[N] has turned to '0' *(newly marked for halting)*, then C='1'; *// set C to continue the while loop iterations*
13.	}
14.	}
15.	Output each node X for which H[X] is "0" as a halting node;
16.	*// End.*

Table 1. Algorithm in pseudo-code

Table 2 shows the node states for this pipeline as this algorithm is applied at each step to simulate a halt of node 9.

Case of example 2 where node 15 halts:
The array H is initialized with a 0 for node 15 and 1's for the rest of the 18 nodes ;
In the 1st iteration of the while loop of line 5:
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
For node 18, the code at line 10 will set H[18] to (H[15] AND H[16])=(0 AND 1)=0;
The flag C will be set to 1 (so that the while loop of line 5 will continue) ;
All other H[] values remain unchanged in this iteration
The 1st iteration of the while loop of line 5 ends and a 2nd one starts because C is '1':
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
For node 16, the code at line 11 will set H[16] to 0;
The flag C will be set to 1 (so that the while loop of line 5 will continue) ;
The 2nd iteration of the while loop of line 5 ends and a 3rd one starts because C is '1':
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
For node 11, the code at line 11 will set H[11] to 0;
For node 12, the code at line 11 will set H[12] to 0;
The flag C will be set to 1 (so that the while loop of line 5 will continue) ;
The 3rd iteration of the while loop of line 5 ends and a 4th one starts because C is '1':
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
For node 5, the code at line 11 will set H[5] to 0;
For node 6, the code at line 11 will set H[6] to 0;
For node 7, the code at line 11 will set H[7] to 0;
For node 8, the code at line 11 will set H[8] to 0;
The flag C will be set to 1 (so that the while loop of line 5 will continue) ;
The 4th iteration of the while loop of line 5 ends and a new one starts because C is '1':
The flag C will be cleared on line 7 ;
In the loop starting at line 8 :
In this iteration there will be no changes in any H[] values, so C will remain cleared ;
The iteration of the while loop of line 5 ends and a new one does not start because C is '0' ;
Program ends with H[] values of 0 indicating the halting nodes.
The halting nodes are those with H[N]=0, which in this run are: 5, 6, 7, 8, 11, 12, 15, 16, 18.

Table 3 shows the node states for this pipeline as this algorithm is applied at each step to simulate a halt of node 15.

As it is evident in the above examples, for the actual coding it is necessary to provide a representation of the pipeline structure on which the code will operate. There are many programming methods to represent data structures such as the forms of the pipelines and most notably variants of linked lists. However, we propose here the use of JSONs as more

Table 2.
Step-by-step node states due to a halt at node 9

		Node State																		
Step	Flag_C	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1.4	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1.13	1	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
2.4	1	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
2.13	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
Result:		0	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

Step	Flag_C	Node State																		
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1.4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
1.13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1
2.4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1
2.13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1
3.4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1
3.13	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1
4.4	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	1	0	1
4.13	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1
5.4	1	1	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1
5.13	0	1	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1
Result:		1	1	1	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0	1

Table 3. Step-by-step node states due to a halt at node 15

portable structured representations, which can be stored, imported and exported easily and provide some relative code independence.

3.5 A JSON supporting structure

A JSON is a programming syntactic concept for storing and exchanging data. It provides a lightweight data-interchange structure that is easy to generate or parse in most programming languages (Python, Java, Javascript, etc.). It is possible to "stringify" a JSON (such a method exists in most languages) for communicating it among servers and clients, as well as to parse it for use within any particular program. The following is one way (out of many possible ones) in which the structure of a pipeline can be represented. This JSON then can be parsed by code that implements the algorithm discussed in section 3.4.

```
{ "Node":
        {
        "NodeID":"1",
        "Inputs":["External"],
        "Outputs":["9"],
        "Dependency": ""
        },
............
        {
        "NodeID":"8",
        "Inputs":["External"],
        "Outputs":["12"],
        "Dependency": ""
        },
        {
        "NodeID":"9",
        "Inputs":["1", "2"],
        "Outputs":["9"],
        "Dependency": "N1*N2"
        },
        {
        "NodeID":"10",
        "Inputs":["3", "4"],
        "Outputs":["13", "14"],
        "Dependency": "N3*N4"
        },
        {
        "NodeID":"11",
        "Inputs":["5", "6"],
        "Outputs":["15", "16"],
        "Dependency": "N5*N6"
        },
............
        {
        "NodeID":"19",
        "Inputs":["17", "18"],
        "Outputs":["External"],
        "Dependency": "N17+N19"
        }
}
```

4. Discussion

One may wonder why would such control be needed since nodes can simply rely on local control according to which, a node simply keeps operating while it is receiving its necessary

inputs and is able to push out its output. It is obviously necessary to keep overall control of the pipeline that will prevent an overall system collapse and probably divert to a possible alternative route to optimize resources. In addition, there might be pipelines with processes (nodes) that push throughput in transient and unstable phase and rely on the subsequent nodes to reach a stable condition (i.e. a node that feeds plastic particles in an injection unit node, which heats and melts plastic before injecting it into the clamping unit node). If the clamping unit is down, then probably the injection unit will continue to keep the plastic in a fluid state until the clamping unit is serviced, but the preceding plastic feeding node should halt to prevent congesting the injection unit with additional plastic particles. Also, this will be the case when two lines of nodes produce components that arrive at the same time and rate at another node to assemble them. If one line halts, the other line should also halt (i.e. in a bottle filling line, one line has multiple nodes that produce caps and stamp them. Another line moves bottles on a conveyor, then fills them with liquid; both lines arrive at a capping station. If the stamping node halts, then the liquid filling line should also halt to prevent placing liquid in open containers exposed for extended time). These examples clearly show that certain nodes might be affected and must halt even if their immediate inputs and outputs are operating normally.

It should be evident however that besides the purposes of control, this method is also valuable for the design of a complex pipeline and for determining the effects of failures and evaluating its fault tolerance.

4.1 Further considerations

In a simple linear pipeline such as the one of Figure 8, the throughput (production rate) of the entire structure is as fast as the slowest stage (node) of the pipeline. This is also the case for a structure like the one of Figure 9 if the dependency of node E is an AND. If the dependency of node E (in Figure 9) is an OR, then the throughput of the segment consisting of nodes E and F has an upper limit to the sum of the throughputs of its two input feeding segments.

In the case of a structure such as the one of Figure 10, the throughput of the segment consisting of nodes A and B has an upper limit to the sum of the throughputs of its two output feeding segments.

Figure 8.
A simple linear pipeline

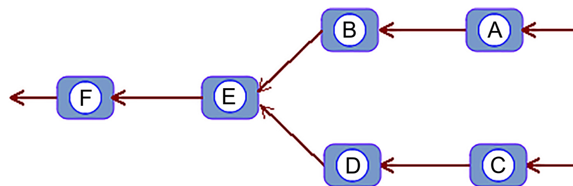

Figure 9.
A pipeline with merging linear segments

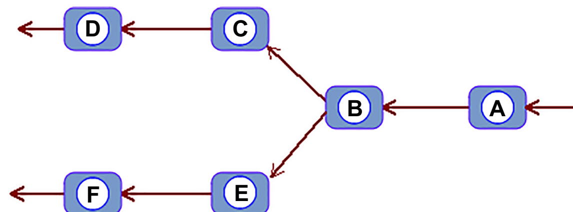

Figure 10.
A pipeline with splitting linear segments

These properties serve to explain that with goals of optimal pipeline designs, it is acceptable to include redundant or repeating substructures. For example, in the case of Figure 9, it is possible that the segments *A-B* and *C-D* be designed to produce the exact same part of which node *E* has two alternative sources (an OR dependency). This would make sense if the production of the part produced by each of these segments is slower than the processing rate through the segment *E-F*. A similar scenario is possible with the structure of Figure 10 where the segments *C-D* and *E-F* may be processing the output part of segment *A-B* in exactly the same way (i.e. they are redundant) because the processing of that part may be much slower than the processing through *A-B*. It follows then that certain redundancies in the design may help both the speed of production and introduce some fault tolerance. On a secondary note, this explains why the dependencies are modeled in the way presented in this study. However, this discussion segment serves more to illustrate why complex designs consisting of substructures like those of Figures 9 and 10 may be needed and why redundant segments may serve purposes of tuning the pipeline throughput as well as purposes of fault tolerance. In such cases, the present work serves to evaluate possible fault scenarios during the design phase of a pipeline, as well as to facilitate its overall control during operation.

With the above concepts in mind, the following would be a way to monitor and control a complex pipeline in which one or more nodes halt.

(1) Each of the pipeline nodes is assumed to include an IoT device that is at least able to run an MQTT client.

(2) One IoT device, designated as the (central) controller, is appointed to run the MQTT broker service. This device may be attached to any of the nodes, or it may be a stand-alone processor (such as a Raspberry Pi, BeagleBone, etc.) that exists only for the purposes of the pipeline control; the only requirement is that it is powerful enough to run the MQTT broker service and the code that implements the checking algorithm (explained in sections 3.4 and 3.5). Running just two topics: EVENT and ACTION, should suffice for our purposes (but it is conceivable that the control may be expanded to more topics as additional purposes may deem appropriate).

(3) All the nodes subscribe as publishers to the topic EVENT and as listeners to the topic ACTION. The EVENT topic is the channel by which new events are communicated by the various nodes. The ACTION topic is the channel by which controls are communicated to nodes.

(4) When a node enters a halting state, it publishes its ID to the EVENT topic to notify the controller about the fact that a new event has occurred, which means that the node is entering or has entered a halting state. For example, if node K halts, its attached IoT device publishes the message “K:Halted” to the EVENT topic. The controller then runs the algorithm discussed previously (section 3.4) to determine what the effect of the current events will be and which nodes will need to also halt. Once the list of other possible nodes (e.g. X, Y, Z) to be halted is determined, this list of node IDs is published in the ACTION topic (e.g. “X:Halt,” “Y:Halt,” “Z:Halt”), and so they are communicated to the nodes. The local controller of each node would receive the published messages and should halt that particular node if it has been tagged for halting (by the main controller).

(5) The same channels can be used to restart a pipeline (or sections of it) as previously halted nodes restart. A node K can publish to the EVENT topic the message “K:Ready.” The controller can run the algorithm to determine which nodes can now restart (and perhaps in what order – but that is an extension to be considered in the future) and accordingly publishes messages in the ACTION topic (e.g. “K:Start,” “X:Start,” “Y:Start,” “Z:Start”).

As a final note, it should be mentioned that to use MQTT, one may choose from a variety of software implementations for it. Some of these software implementations are very plain and simple such as *MOSQUITTO* (Eclipse Mosquitto™, 2020), and some are more advanced such as *MQTT SPARKPLUG* (Obermaier, 2020) and (Eclipse Foundation, 2019), which standardizes the communication data format and its interpretation. It also allows communication with non-MQTT devices or from using data from other protocols such as OPC-UA or Modbus. One also must be aware of possible security concerns as is usually the case with networked systems. Specifically, in an MQTT implementation where the subscription for publishing is open (i.e. the Broker accepts subscriptions without authentication), it is possible for any agent to subscribe to the messaging topics and create havoc. So, users should be aware of various methods to address security issues such as client authentication, payload encryption, use of TLS transport and the vulnerabilities of the various network layers. Some informative articles can be found in MQTT Security Fundamentals (2015) by the HiveMQ Team and in Dinculeană and Cheng (2019) among many others available.

5. Conclusion

Referring to pipeline designs, the present framework and analysis method can be used to determine "what if" scenarios of node failure(s) and thus evaluate the fault tolerance properties of a pipeline proposed design and possibly guide its development. It is also possible to extend this work to consider expected probabilities of failures of various nodes (which could possibly reflect statistical data of node components) to assess a design and drive updates.

References

Bierbooms, R. (2012), *Performance Analysis of Production Lines: Discrete and Continuous Flow Models*, Technische Universiteit Eindhoven, Eindhoven.

Cerrada, M., Cardillo, J., Aguilar, J. and Faneite, R. (2007), "Agents-based design for fault management systems in industrial processes", *Computers in Industry*, Vol. 58, pp. 313-328.

Dal, B. (1999), *Audit and Review of Manufacturing Performance Measures at Airbags International Limited*, UMIST.

Dal, B., Tugwell, P. and Greatbanks, R. (2000), "Overall equipment effectiveness as a measure of operational improvement: a practical analysis", *International Journal of Operations and Production Management*, Vol. 20 No. 12, pp. 1488-1502.

Dinculeană, D. and Cheng, X. (2019), "Vulnerabilities and limitations of MQTT protocol used between IoT devices", *Applied Sciences*, Vol. 9 No. 5, 848.

Eastburn, J. (2020), "How MQTT is advancing automation and control", Process Instrumentation, available at: https://www.piprocessinstrumentation.com/process-control-automation/article/21122615/how-mqtt-is-advancing-automation-and-control#:~:text=MQTT%20creates%20an%20efficient%20data,quality%20reported%20in%20real%2Dtime.

Eclipse Foundation (2019), "Sparkplug MQTT topic and payload specification rev 2.2", available at: https://www.eclipse.org/tahu/spec/Sparkplug%20Topic%20Namespace%20and%20State%20ManagementV2.2-with%20appendix%20B%20format%20-%20Eclipse.pdf.

Gao, S., Rubrico, J.I., Higashi, T., Kobayashi, T., Taneda, K. and Ota, J. (2019), "Efficient throughput analysis of production lines based on modular queues", *IEEE Access*, Vol. 7, pp. 95314-95326, doi: 10.1109/access.2019.2928309.

Glasserman, P. and Yao, D.D. (1994), "A GSMP framework for the analysis of production lines", in Research, S.S. (Ed.), *Stochastic Modeling and Analysis of Manufacturing Systems*, Springer, New York, NY, doi: 10.1007/978-1-4612-2670-3_4.

Hasan, H. and Mohammad, B. (2018), "Evaluation of MQTT protocol for IoT based industrial automation", *IJESC*, Vol. 8 No. 12, pp. 19364-19369.

Hechtman, S. (2021), ISA Interchange, available at: https://blog.isa.org/mqtt-ideal-connectivity-protocol-industrial-internet-of-things.

Jaloudi, S. (2019), "Communication protocols of an industrial internet of things environment: a comparative study", *Future Internet*, Vol. 11 No. 3, 66.

Landers, R., Barton, K., Devasia, S., Kurfess, T., Pagilla, P. and Tomuzuka, M. (2020), "A review of manufacturing process control", *Journal of Manufacturing Science and Engineering*, Vol. 142 No. 11, p. 23, 110814.

Lee, E. (2006), "Cyber-physical systems - are computing foundations adequate?", *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*.

Lee, J., Ardakani, H., Yang, S. and Bagheri, B. (2015a), "Industrial big data analytics and cyber-physical systems for future maintenance and service innovation", *The Fourth International Conference on Through-life Engineering Services*, Cranfield, pp. 3-7.

Lee, J., Bagheri, B. and Kao, H.-A. (2015b), "A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems", *Manufacturing Letters*, Vol. 3, pp. 18-23.

Mehmood, F., Ahmad, S. and Kim, D. (2019), "Design and implementation of automation appliances control based on MVC model using distributed MQTT broker in CoT networks", *International Journal of Innovative Technology and Exploring Engineering*, Vol. 8 No. 3C, pp. 262-269.

Monostori, L., Bauernhansl, T., Kádár, B. and Kondoh, S. (2016), "Cyber-physical systems in manufacturing", *CIRP Annals*, Vol. 65 No. 2, pp. 621-641.

Mosquitto, E. (2020), "An open source MQTT broker", Mosquitto, available at: https://mosquitto.org/.

MQTT (2019), "In Wikipedia", available at: https://en.wikipedia.org/wiki/MQTT, March 7.

MQTT (2021), "MQTT", METT: The Standard for IoT Messaging, available at: https://mqtt.org/.

MQTT Security Fundamentals (2015), HiveMQ, available at: https://www.hivemq.com/blog/mqtt-security-fundamentals-wrap-up/.

Obermaier, D. (2020), "MQTT sparkplug essentials", HIVEMQ, available at: https://www.hivemq.com/blog/mqtt-sparkplug-essentials-part-1-introduction/.

Parente, M., Figueira, G., Amorim, P. and Marquez, A. (2020), "Production scheduling in the context of Industry 4.0: review and trends", *International Journal of Production Research*, Vol. 58 No. 17, pp. 5401-5431.

Phadnis, V.S. (2013), "Production line design and system analysis for new product", Master Thesis, MIT, available at: http://hdl.handle.net/1721.1/85788.

Starkov, K.K., Feoktistova, V., Pogromsky, A.Y., Matveev, A. and Rooda, J.E. (2012), "Performance analysis of a manufacturing line operated under optimal surplus-based", *Mathematical Problems in Engineering*. doi: 10.1155/2012/602094.

Syafrudin, M., Alfian, G., Fitriyani, N. and Rhee, J. (2018), "Performance analysis of IoT-based sensor, big data processing, and machine learning model for real-time monitoring system in automotive manufacturing", *Sensors*, Vol. 18 No. 9, 2496.

Wang, L. (2013), "Machine availability monitoring and machining process planning", *CIRP Journal of Manufacturing Science and Technology*, Vol. 6, pp. 263-273.

Wang, L., Törngren, M. and Onori, M. (2015), "Current status and advancement of cyber-physical systems in manufacturing", *Journal of Manufacturing Systems*, Vol. 37 No. 2, pp. 517-527.

Corresponding author
Mohammad Saadeh can be contacted at: msaadeh@selu.edu

For instructions on how to order reprints of this article, please visit our website:
www.emeraldgrouppublishing.com/licensing/reprints.htm
Or contact us for further details: **permissions@emeraldinsight.com**