Software-defined Environment for Reconfigurable Manufacturing Systems

Naresh Ganesh Nayak, Frank Dürr, Kurt Rothermel
Institute for Parallel and Distributed Systems
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart
Email: {naresh.nayak, frank.duerr, kurt.rothermel}@ipvs.uni-stuttgart.de

***Abstract*—Conventional manufacturing systems like assembly lines cannot handle the constantly changing requirements of a modern-day manufacturer, which are driven by volatile market demands. In a bid to satisfy such requirements, modern manufacturing systems, comprising innumerable cyber-physical systems (CPS), aim to be reconfigurable. CPS implement production processes through an ICT infrastructure networked with sensors and actuators embedded in the shop floor. Reconfigurability, in context of manufacturing systems, must include the entire system of networked components and hence requires a flexible ICT infrastructure. Providing flexible ICT infrastructures, often, comes at the cost of diluted quality of service (QoS) guarantees. This, however, is not an option for manufacturing systems, most of which require strict QoS guarantees to function correctly. To overcome this obstacle, we propose a new software-defined environment (SDE) for reconfigurable manufacturing systems with real-time properties in this paper. Software-defined environment is an emerging technology that provides flexible ICT infrastructures modifiable using software. Our contributions include an SDE-based system architecture for dynamically configuring the underlying infrastructure for a manufacturing system. In particular, we focus on configuring the network for the time-sensitive communication flows essential for realising CPS. Moreover, we propose a pair of routing algorithms to calculate routes for these flows while configuring the network.**

Keywords*—*Time-sensitive networks, Industry 4.0, Quality of service, Genetic Algorithm, Software-defined networks

I. Introduction

Gone are the days when manufacturers targeted producing standardized goods with increased production rates. In this modern era, they are forced to offer higher degrees of customization in their products to retain their competitive edge. Daimler, a leading automotive manufacturer, offers around 10^{24} variants of its Mercedes E-Class model [1]. Further, there is a dramatic decrease in the average product life-cycle to the extent that it is no longer viable to design a dedicated manufacturing system for a specific product. For instance, in the fiercely competitive automotive market, the average product life-cycle for cars has reduced from 10.6 yrs in the 1970's to 5.6 yrs in the early 2000's [2]. In these modern scenarios, manufacturers seek to produce highly customized products at mass production costs. Moreover, they expect their manufacturing facilities to turnaround in a short timespan to produce new products or variants. Existing systems like the assembly lines etc., fall short of these expectations.

Smart factories, as envisioned by the Industry 4.0 initiative, adapt themselves to volatile environments for satisfying the requirements of a modern-day manufacturer [3]. For this, they use manufacturing systems that can be reconfigured rapidly. Such manufacturing systems can adjust their production capacities and processing functionalities to respond in-time towards changing market demands, regulations etc. They also support various kinds of production processes — the sequence of steps involved in manufacturing a product — albeit with some reconfiguration effort.

Reconfigurability in manufacturing systems not only applies to the mechanical ("physical") components of a factory such as robots and machines, but also to its information and communication technologies (ICT) infrastructure consisting of compute, storage, and network resources. Modern manufacturing systems are, in fact, cyber-physical systems (CPS) that control production processes through an ICT infrastructure. In these systems, CPS controllers sense the state of the production processes through a set of sensors and manipulate this state through actuators. Often, these components are distributed physically across the shop floor and communicate through a communication network. Thus, reconfigurability must comprise the entire system of networked components and, hence, requires a flexible ICT infrastructure.

A. Challenges

Why is it hard to provide a flexible ICT infrastructure for manufacturing? Most manufacturing systems require strict quality of service (QoS) in order to function correctly. In particular, this applies to the real-time system behaviour since most of these systems control time-sensitive production processes. Providing QoS is typically much easier if a dedicated infrastructure is provided, for instance, dedicated computers hosting the CPS controllers. Another typical example, which we will focus on in this paper, are real-time communication networks. Often, such networks are made from dedicated field-buses that provide real-time guarantees for communication.

Although dedicated components can provide the required QoS more easily, they often do not meet the flexibility requirements of reconfigurable systems. For instance, changing the hardware configuration (sensors, actuators) might require cumbersome "re-wiring" of the field-bus network. Therefore, we strive for a system that can provide flexibility without sacrificing essential QoS properties.

In order to keep our discussions and solutions focused, we will focus on the networking domain in the following.

Thus, our concrete goal is to develop methods for flexible time-sensitive networking suitable for reconfigurable manufacturing systems.

B. Proposed Solution

To improve flexibility of manufacturing systems, we propose a *software-defined environment*, inspired from the basic principles of software-defined networking (SDN), targeted at manufacturing scenarios [4]. The primary goal of SDN is to increase the flexibility of networking. To this end, SDN clearly separates the basic network functionality of forwarding (network data plane) from the network configuration (network control plane). The network data plane is implemented "in hardware" by network switches, whereas the network control plane is outsourced to standard hosts implementing the logic to configure the network, e.g., the forwarding tables of switches. By implementing an application-specific network control logic, we can shape the network to meet the requirements of the application. This also enables the dynamic reconfiguration of the network according to the requirements of the reconfigurable manufacturing system, for instance, by configuring suitable paths between a dynamic set of sensors and actuators.

Although this concept in general improves flexibility, it is an open question how to provide the desired QoS to the manufacturing systems through suitable network control logic. So far, the concepts for SDN and time-sensitive networking (TSN) — for instance, proposed in the IEEE 802.1 standard [5] — have been developed independently of each other. Thus, the major question is, can we combine the benefits of SDN and TSN to build a time-sensitive software-defined network (TSSDN)? Such a TSSDN would become one cornerstone of a software-defined environment for manufacturing, which can then be complemented by additional concepts for the compute and storage domain.

In this paper, we make the first step towards such a "software-defined factory" by making the following contributions:

- We present an *architecture of a software-defined environment* for reconfigurable manufacturing systems. In particular, we focus on time-sensitive software-defined networking to support real-time communications in manufacturing systems.
- We introduce a *specific routing problem for TSSDN* and present *heuristic-based routing algorithms* for tackling it.
- Our evaluations demonstrate the effectiveness of software-defined environments for handling time-sensitive communication flows in manufacturing systems.

The rest of the paper is structured as follows. First, we discuss related work in Section II. Section III introduces the operating principles of a software-defined approach and presents our architecture of software-defined environment for reconfigurable manufacturing systems. We present a problem related to routing of time-sensitive communication flows in manufacturing systems and a set of algorithms to solve it in Section IV. We evaluate these algorithms in Section V followed by a short discussion in Section VI. Finally, we conclude and present the future outlook for our work in Section VII.

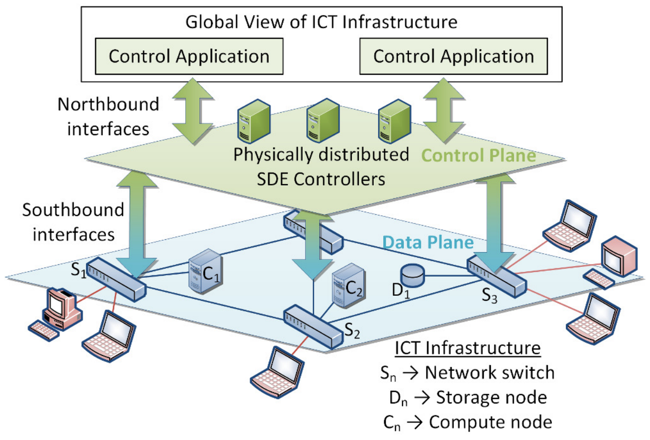


Fig. 1: Basic principles of software-defined environments — separation of data plane and control plane, logical centralization of control plane providing a global view onto the infrastructure

II. Related Work

Though reconfigurable manufacturing systems are seen as a central feature of smart factories, their exact design is currently a subject of research. Various issues involved in the design of reconfigurable manufacturing systems have been discussed in the literature [6], [7], [8]. However, these are focused only on aspects of mechanical design for making manufacturing systems reconfigurable. Despite an indispensable role of ICT infrastructures in modern manufacturing, not much has been done to incorporate reconfigurability in it.

With respect to communication networks, there exists a wealth of literature on mechanisms that provide QoS guarantees for communication in manufacturing environments [9], [10]. However, most of these mechanisms use field-bus technologies which are not flexible while others provide QoS guarantees only in static scenarios.

Our vision of the software-defined factory fills these research gaps by creating highly flexible ICT infrastructures for reconfigurable manufacturing systems that provide the desired QoS guarantees for the manufacturing systems, especially with respect to the networking domain.

III. Software-defined Environment for Manufacturing Systems

As already mentioned in Section I, one of the major requirements for reconfigurable manufacturing systems is a flexible ICT infrastructure that can be reconfigured easily to follow the dynamic setup of sensors, actuators, and CPS controllers deployed on the shop floor. Fixed infrastructures with statically assigned dedicated hardware cannot provide the required flexibility. Instead, we strive for a "software-defined" environment, where the hardware configuration can be adapted to the dynamic configuration of the manufacturing system.

The two basic operating principles of software-defined environments are shown in Figure 1. The first of these principles is the *separation of data plane and control plane*. These terms, as already explained in Section I, originate from the networking domain. However, they can also be translated to the compute and storage domains by separating their core functionality (processing and storing data respectively) from the control and configuration of the corresponding systems. For instance, managing (creating, migrating, terminating, etc.) virtual machines can be seen as a typical task of the control plane of a software-defined compute system. The software process implementing the control plane is called the SDN controller or more generally SDE (software-defined environment) controller if we do not restrict ourselves to network resources. Typically, this SDE controller offers two interfaces: a southbound interface between itself and the underlying infrastructure resources like network switches or hosts, and a northbound interface towards the SDE control applications that implement control logic for specific management tasks like a routing algorithm for configuring switches.

The second operating principle of our envisioned software-defined environment is *logically centralized control* to facilitate the implementation of control logic. Basically, logical centralization provides a *global view* onto the infrastructure to the SDE control applications, which simplifies the implementation of control logic significantly. For instance, an SDN controller can implement improved routing algorithms with a global view onto the whole network including network topology, traffic (link loads), etc. Thus, instead of implementing a distributed routing algorithm, we can simply use a centralized algorithm. Of course, logical centralization does not imply that the SDE controller is also physically centralized. Usually, it is physically distributed to several hosts to increase availability and performance.

Applying these two basic principles to a software-defined environment for a reconfigurable manufacturing system means that we outsource the control of the manufacturing system to a logically centralized SDE controller. This provides it with a global view onto the whole infrastructure including all sensors, actuators, network switches, compute resources, storage resources along with applications like the constituting CPS controllers. The SDE controller can then configure the infrastructure based on its holistic view. In order to adapt to dynamic changes, which is essential in a reconfigurable system, the SDE controller implements a so-called MAPE (Monitor, Analyze, Plan & Execute) loop as shown in Figure 2. The tasks executed by the SDE controller as a part of the MAPE loop are enumerated as follows:

- **Monitoring** - In the monitoring phase, the SDE controller gathers the state of the infrastructure for generating the aforementioned global view. For this, it measures performance metrics like network latency, computing load, system throughput, etc., in the data plane. State-of-the-art techniques which utilise the south-bound interfaces could be readily used for estimating these performance metrics [11].

- **Analysis** - The state of the infrastructure gathered in the monitoring phase is analysed by the SDE controller. Advanced techniques using machine learning, time series modelling etc. may be used for this purpose.

- **Planning** - In the planning phase, the SDE controller uses the information inferred from the analysis phase and plans suitable reconfiguration steps by implementing centralized control logic. For instance, the controller can use the state obtained from monitoring to implement a traffic-aware adaptive routing algorithm for the underlying network.

- **Execution** - In this final phase, the SDE controller executes the planned reconfigurations through commands via the southbound interfaces. An example of such reconfiguration would be modifications to the forwarding tables in the network switches.

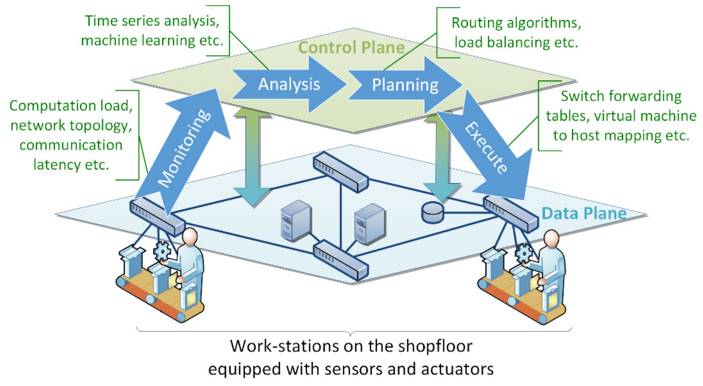

Time series analysis, machine learning etc.
Routing algorithms, load balancing etc.
Control Plane
Analysis
Planning
Computation load, network topology, communication latency etc.
Monitoring
Execute
Switch forwarding tables, virtual machine to host mapping etc.
Data Plane
Work-stations on the shopfloor equipped with sensors and actuators

Fig. 2: MAPE loop in the control plane of the software-defined environment for Reconfigurable Manufacturing Systems

One specific requirement of an SDE for manufacturing systems, most of which are highly time-sensitive in nature, is to configure the infrastructure in such a way that time-bounds of these systems are fulfilled, e.g., guaranteed upper-bounds on communication delay between sensors or actuators and the CPS controller. The basic principle of logically centralized control also facilitates the control of time-sensitive systems. For instance, as we will show in this paper in detail, by knowing all applications (data sources and sinks), application requirements (time bounds), and by having control over all network resources (switches), we can configure network paths through a logically centralized routing algorithm such that network flows do not interfere and time-bounds are met.

Thus, a general challenge is to implement control logic that provides desired QoS to the time-sensitive manufacturing systems. In this paper, we will focus on control algorithms for time-sensitive software-defined networking, although the problem also applies to other resources like the scheduling of computation processes.

IV. Routing Time-Sensitive Communication Flows

In this section, we present a problem related to routing of time-sensitive communication flows constituting the CPS in a factory along with a basic algorithm to solve it. Such algorithms play a role in the planning phase of the MAPE loop executed by the SDE controller. We further introduce a pair of heuristics to guide this basic algorithm towards improved solutions.

Generally, CPS require varying numbers of time-sensitive communication flows between its components based on the type of production process it implements and the number of components it has on the shop floor. For instance, CPS responsible for open-loop production processes typically require lower number of time-sensitive flows compared to the ones responsible for closed-loop production processes. These time-sensitive flows have stringent QoS requirements, in particular the strict bounds they place on acceptable end-to-end network delay and corresponding jitter [9]. The network delay comprise propagation delay, processing delay, and queuing delay [12]. Propagation delay is the time required for a data packet to propagate through the network links, processing delay is the time required for the network switches to process and forward the packet, while queuing delay is the time that the packet spends in the queues of the network switches. Of these delays, propagation delay (hundreds of nanoseconds) and processing delay (few microseconds) are negligible compared to the queuing delay (couple of milliseconds). Thus, a significant part of the end-to-end delay results from the queuing delay which occurs mainly due to in-network congestion. Fluctuating queuing delays result in jitter. Hence, while routing time-sensitive flows, a routing algorithm must additionally consider queuing effects in network switches, which is otherwise not required.

If network links are exclusively allocated to communication flows, queuing delays can be eliminated bounding end-to-end delays along with jitter [13]. Hence, to provide the desired QoS guarantees to these time-sensitive flows, we intend to exploit the path diversity in the network and allocate an edge-disjoint path (paths that do not share network links) to each of these flows. The allocated routes are then programmed into the network by the SDE controller in the execution phase of the MAPE loop. In case of over-subscription of network resources, we aim to maximise the number of CPS that have edge-disjoint paths for all their flows. Such routing algorithms execute on the control plane of the software-defined environment and can, hence, benefit from the global view of the infrastructure along with the knowledge of the CPS requirements with respect to its time-sensitive flows. The following subsections present the system model and the formal problem statement followed by the routing algorithms.

A. System Model

In our system model, we address the underlying network for the CPS along with their time-sensitive flows. The network is modelled as a directed graph, $G \equiv (V, E)$, where V is the set of network nodes and E is the set of edges representing the network links connecting the nodes. Further, $V \equiv S \cup H$, where S is the set of network switches and H is the set of hosts in the network. Also, $E \subseteq V \times V$ such that if (v_1, v_2) $\in E$, then (v_2, v_1) $\in E$. This models the full-duplex nature of switched Ethernet used in software-defined environments.

$CPSSet$ represents the set of target CPS required for implementing the production processes in the shop floor. For $cps \in CPSSet$, cps = $\{(h_i, h_j) \mid h_i, h_j \in H\}$, i.e., a cyber-physical system is described as a set of tuples, each containing a pair of hosts — the source host and the destination host for a time-sensitive flow.

We refer to a CPS as "realised" if the used routing algorithm assigns network paths for all its time-sensitive flows. We model the solution to this routing problem of time-sensitive flows as a tuple — ($CPSSeq$, $FlowSeqMap$). $CPSSeq$ is a sequence of the target CPS, $CPSSeq \equiv [cps_1, cps_2, \dots cps_n]$. While, $FlowSeqMap$ is a map that gives the sequence of the flows corresponding to a CPS. i.e., $FlowSeqMap[cps_i] \equiv [f_{i,1}, f_{i,2}, \dots f_{i,j}]$, where $f_{i,j}$ is the j^{th} flow of system cps_i. Further, we define the fitness (quality) of a given solution as the number of CPS that are realised by the basic routing algorithm described below.

B. Problem Statement

Our routing approach allocates edge-disjoint paths to the time-sensitive flows while seeking to maximise the number of CPS so realised. Edge-disjoint paths avoid the problem of competing flows along any path, thus, eliminating the problem of network congestion and high queueing delay. Note, that in this paper we do not strive to allocate several flows to the same link. Although this could further increase throughput and number of supported CPS, it requires more complex solutions including scheduling mechanisms at switches, which are subject to future work.

Merely maximising the number of realised CPS raises fairness issues for those requiring higher number of flows. We can mitigate this issue by allocating weights to each CPS based on their importance and then maximise the sum of weights for the set of realised CPS. While our algorithms can be easily extended for this, it is currently out of scope for this paper.

C. Calculating edge-disjoint routes for time-sensitive communication flows

Given a graph G along with a set $T \equiv \{(s_1, d_1), (s_2, d_2), \dots (s_n, d_n)\}$ containing pairs of hosts in the networks, the problem of maximising the number of pairs to which edge-disjoint paths can be allocated is known as the maximum edge-disjoint paths problem. This well-known problem is NP-Hard and requires heuristic approaches for finding solutions that are close to the optimal ones [14]. Our problem formulation takes the maximum edge-disjoint paths problem a step further. Instead of maximising the number of flows for which edge-disjoint paths can be allocated, we seek to maximise the number of CPS that are realised as a result. Our problem however reduces to the maximum edge-disjoint paths problem, if all the CPS in $CPSSet$ consist of only one time-sensitive flow.

Algorithm 1 describes the basic routing algorithm to allocate edge-disjoint paths to the time-sensitive flows based on an input solution described by the tuple ($CPSSeq$, $FlowSeqMap$).

The algorithm realises the CPS as per the order dictated by the $CPSSeq$ (Line 2). For realising a CPS, the algorithm calculates and allocates routes to all its flows in the order defined by the map $FlowSeqMap$ indexed by the corresponding CPS (Line 3). The routes for the flows are determined by executing Dijkstra's algorithm (described in [15]) on graph G (Line 4).

Algorithm 1 Basic algorithm to route time-sensitive flows

```
Require: Graph $G$, Solution ($CPSSeq$, $FlowSeqMap$)
 1: $Routes \leftarrow \{\ \}$
 2: for $cps$ in $CPSSeq$ do
 3:     for $flow$ in $FlowSeqMap[cps]$ do
 4:         $flowPath \leftarrow$ Dijkstra($G, flow.src, flow.dst$)
 5:         if $flowPath$ != NULL then
 6:             $Routes[flow] \leftarrow flowPath$
 7:             $G.edges \leftarrow G.edges - flowPath$
 8:         else
 9:             deallocate all the assigned flows of $cps$
10:             add corresponding links back to $G$
11: return $Routes$
```

To ensure that a network link is not allocated to multiple flows, we remove the corresponding edge from the graph G when it is allocated to a flow (Line 7). If no route can be found for some flow, then the corresponding CPS cannot be realised. It is then futile to allocate the expensive network resources for its other flows. Hence, we deallocate such flows and return the respective network links to the graph G (Line 9–10).

The fitness of a solution with this basic algorithm depends on the ordering of the CPS in $CPSSeq$ and the corresponding ordering of flows in $FlowSeqMap$. In the following, we improve the basic algorithm to derive a greedy algorithm (heuristic approach) and a genetic algorithm (meta-heuristic approach) that generate better solutions, i.e., increase the number of realised CPS.

1) Greedy Algorithm: In the greedy algorithm, we generate a few candidate solutions heuristically and then select the best one from them. This algorithm takes four inputs: i) G, the network graph, ii) $CPSSet$, the set of target CPS, iii) $MaxSolns$, the number of candidate solutions to be considered, iv) $FlowSeqMap$, a map that gives the initial ordering of the flows corresponding to a CPS.

Algorithm 2 Greedy algorithm for maximising the number of realised CPS

```
Require: Graph $G$, Set of CPS $CPSSet$, Flow Sequence Map
    $FlowSeqMap$, No. of candidate solns $MaxSolns$,
 1: $CandidateSolutions \leftarrow [\ ]$
 2: $BestSolution \leftarrow$ NULL; $BestSolnFitness \leftarrow 0$
 3: for $i = 0$ to $MaxSolns$ do
 4:     $CPSSeq \leftarrow$ stable_sort(shuffle($CPSSet$))
 5:     $CandidateSolutions.add((CPSSeq, FlowSeqMap))$
 6: for $Soln$ in $CandidateSolutions$ do
 7:     $SolnFitness \leftarrow$ calculate_fitness($G$, $soln$)
 8:     if $SolnFitness > BestSolnFitness$ then
 9:         $BestSolnFitness \leftarrow SolnFitness$
10:         $BestSolution \leftarrow Soln$
11: return $BestSolution$
```

In the greedy algorithm, described in Algorithm 2, we use the number of flows required to realise a CPS as the heuristic. We create different sequences of target CPS in which they are sorted in an ascending order of the number of constituent flows, i.e., $CPSSeq_i$, where $i \equiv \{1, 2, \dots MaxSolns\}$. Each of these sequences is combined with the default ordering of flows given by $FlowSeqMap$ to generate $MaxSolns$ candidate solutions, i.e., ($CPSSeq_i$, $FlowSeqMap$). To allow these candidate solutions to vary significantly, we generated $CPSSeq_i$ by randomly shuffling the set of target CPS, $CPSSet$, and then executing a sorting process (Line 4). Since we used a stable sort, the relative order of the CPS before sort is maintained after sort as well. We select the best of the candidate solutions after evaluating their individual fitness (Line 6–8). Function calculate_fitness() (Line 7) (essentially similar to the Algorithm 1) tracks and returns the number of CPS realised by the solution using the basic routing algorithm.

The greedy algorithm uses the default ordering of flows, as given in the $FlowSeqMap$, due to lack of reliable heuristics for ordering the flows of a particular CPS for routing. To explore the solution space effectively, we developed a meta-heuristic approach using a genetic algorithm that also varies the ordering of flows in $FlowSeqMap$.

2) Genetic Algorithm: Typically, the first step in genetic algorithms is to generate a set of candidate solutions, known as the population. The quality (fitness) of the population is then iteratively improved by execution of three genetic operators — Selection, Cross-over and Mutation. The genetic algorithm described in Algorithm 3 needs an additional input as compared to the greedy algorithm: $MaxIterations$, the number of iterations/generations for which the candidate solutions are allowed to evolve.

We generated the initial population using the method similar to the one used in the Greedy Algorithm, i.e., using stable sort after random shuffling of $CPSSet$ (Line 3). Additionally, we randomly shuffled the ordering of flows in $FlowSeqMap$ corresponding to every CPS (Line 4–5) for each candidate solution. Further, we improved the calculation of solution fitness to break ties between candidates that realise an equal number of CPS. In such cases, the fitness calculation procedure takes into account the number of flows set up for breaking the tie. The higher the number of flows set up, the better the solution.

Algorithm 3 Genetic algorithm for maximising the number of realised CPS

```
Require: Graph $G$, Set of CPS $CPSSet$, Flow Sequence Map
    $FlowSeqMap$, No. of candidate solns $MaxSolns$, No. of
    iterations $MaxIterations$,
 1: $Population \leftarrow [\ ]$
 2: for $i = 0$ to $MaxSolns$ do
 3:     $CPSSeq \leftarrow$ stable_sort(shuffle($CPSSet$))
 4:     for $cs$ in $CPSSet$ do
 5:         $FlowSeqMap[cs]$ = shuffle($FlowSeqMap[cs]$)
 6:     $Population.add((CPSSeq, FlowSeqMap))$
 7: for $i = 0$ to $MaxIterations$ do
 8:     $SelectPopulation \leftarrow$ selection($G$, $Population$)
 9:     $NextGeneration \leftarrow$ crossover($SelectPopulation$)
10:     $Population \leftarrow$ mutate($NextGenPopulation$)
11: return fittest($Population$)
```

With each iteration, the population evolves with improving quality of solutions. In each iteration, three genetic operators (Line 8–10) are applied on the population:

1) *Selection* is responsible for selecting the candidate solutions from the current generation that contribute towards the candidate solutions for the next generation. We used roulette wheel selection mechanism for this operator [16]. With this method, candidate solutions with higher fitness have a higher probability of making the cut while solutions with lower fitness head for extinction. This operator, hence, needs graph G as input for calculating the fitness of the population.
2) *Cross-over* combines two "selected" candidate solutions of the current generation to generate two solutions for the next generation. This operation must be carefully designed to ensure that the candidate solutions for the next generation have a high probability of improving over their parents from the current generation. We used a uniform cross-over method for this operation [16]. This implies that two parent solutions are combined to construct two child solutions, such that the children get approximately half of the solution ($CPSSeq$ as well as $FlowSeqMap$) from each parent. Further, we also used elitist selection method that allows the fittest solution to walk into the next generation unaltered.
3) *Mutation* alters candidate solutions slightly to maintain diverse solutions for a wider exploration of the solution space. Typically, the mutation operator is applied with a very low probability to avoid randomizing the candidate solutions. While applying this operation, we mutated randomly selected candidate solutions by swapping a randomly selected pair of systems in its $CPSSeq$.

At the end of the specified iterations, we choose the fittest solution present in the population (Line 11).

V. Evaluations

Our algorithms route the time-sensitive communication flows over edge-disjoint paths. The resulting end-to-end delay is then the sum of propagation delay of the network links and the processing delay of the network switches over which the flow is routed. The queuing delay is eliminated as no two flows contend for access to any network link. The evaluation results in [12] found the processing delay of a state-of-the-art SDN switch to be constant (around 3.8μs). Further, the propagation delay in the corresponding 1 Gbps link was estimated to be constant (around 5–15ns) depending on its length. Thus, using edge-disjoint paths for routing results in minimal end-to-end delays (in the microseconds range) for the time-sensitive flows.

In the following, we also evaluated our algorithms in terms of the quality of the solution that they generate, i.e., the number of CPS that they realise, and their runtimes. For this, we simulated them to calculate the edge-disjoint network paths for time-sensitive flows of a varying set of CPS over random network topologies generated using Erdős-Rényi model [17]. We used NetworkX to generate these random topologies for simulating the evaluation scenarios [18].

The algorithms were primarily evaluated in two phases. In the first phase, we compared the quality of solution with the optimal solution for a small topology. Note that due to the high complexity of the problem, such a comparison with the optimum is only possible for smaller scenarios. The goal here was to gauge if the solutions generated by our algorithms are close enough to the optimal solutions. In the second phase, we compared the performance of the algorithms with each other when executed on larger problem sizes. Here, we evaluated the runtime of the algorithms along with the quality of solutions generated in each of the evaluation scenario.

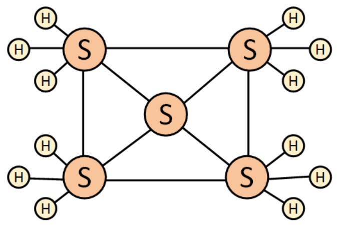


Fig. 3: Small topology for benchmarking. S indicates the network switches while H indicates the end-hosts.

Algorithms	Category	Mean	Std. Dev	Optimal Solution
Genetic Algorithm	Systems Realised	6.42	0.49	42 times
	Flows	9.95	1.81	42 times
Greedy Algorithm	Systems Realised	6.16	0.37	16 times
	Flows	8.76	1.61	16 times

TABLE I: Results of 100 execution runs of greedy and genetic algorithm on the benchmark topology shown in Figure 3.

A. *Performance Comparison with Optimum for Small Problem Sizes*

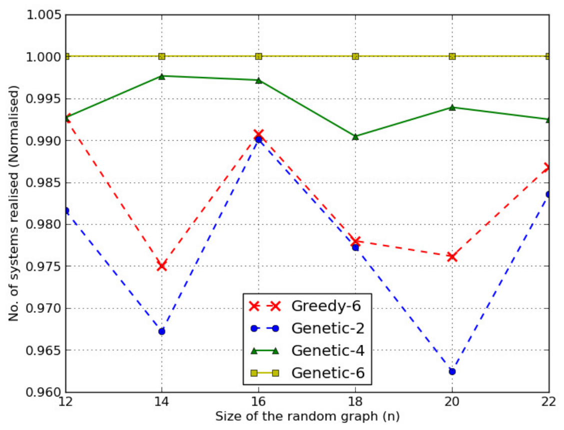
To determine if the solutions generated by our algorithms are close enough to the optimal solution, we created a small benchmark topology with high path diversity consisting of 5 network switches and 12 hosts, as shown in Figure 3. Further, we created a set of 20 CPS, each requiring between 1 to 5 time-sensitive flows. By exhaustively searching the solution space, we determined that the optimal solution for this problem realises 7 CPS consisting of 12 flows in total.

We executed the greedy and the genetic algorithms on the benchmark topology with inputs to consider 6 candidate solutions. We allowed the genetic algorithm to perform 4 iterations to improve the candidate solutions. The results of 100 execution runs of these algorithms are summarised in Table I. For these executions, the solutions generated by the genetic and the greedy algorithm could, on an average, realise 6.42 and 6.16 CPS respectively compared to the 7 CPS realisable using the optimal solution. Moreover, the genetic algorithm could produce the optimal solution 42 times out of the 100 execution runs.

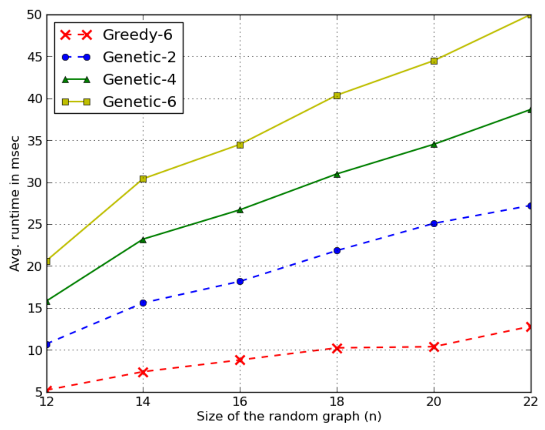
Thus, for smaller topologies, the solutions generated by our algorithms are quite close to the optimal ones. Further, they are able to generate the optimal solutions frequently in case they are executed multiple times.

B. *Comparison of algorithms*

In the second phase of evaluations, we compared the performance of our algorithms with each other. For this, we



(a) Quality of solutions produced



(b) Runtime of the algorithms

Fig. 4: Execution results (Average of 100 execution runs) of genetic algorithm and greedy algorithm considering varying number of candidate solutions.

n	Genetic Algorithm			Greedy Algorithm		
	Systems Realised	Flows	Runtime	Systems Realised	Flow	Runtime
30	29.72	54.2	105ms	29.04	52.25	18ms
40	39.36	70.45	138ms	38.87	63.48	24.3ms
50	49.43	91.49	188ms	48.8	90.07	34.2ms

TABLE II: Average results of 100 execution runs of greedy and genetic algorithm on random topologies generated using Erdős-Rényi model (p = 0.25 and varying n).

created random graphs using the Erdős-Rényi model, denoted as $G(n, p)$ [17]. These graphs consist of n nodes with p denoting the probability that any two nodes are connected by an edge. We also created a set of target CPS containing $2n$ systems, each consisting of between 1 to 3 (uniformly distributed) time sensitive flows. We executed our algorithms on these randomly generated graphs and compared their performance with respect to the quality of the solutions they generated and their corresponding runtimes.

To ensure that none of these algorithms gain an undue advantage, we executed both the algorithms with the input to consider only 6 candidate solutions. The genetic algorithm was executed to perform 4 iterations.

The Table II summarises the results of 100 execution runs of the algorithms on random topologies generated using the Erdős-Rényi model with varying n. These results show that when both the algorithms consider an equal number of candidate solutions, the average solution provided by the genetic algorithm is better than the greedy algorithm although its runtime is an order of magnitude higher than that of the greedy algorithm.

Finally, we also executed the algorithms to evaluate if the genetic algorithm can outperform the greedy algorithm despite considering a lower number of candidate solutions, thereby decreasing the penalty in runtime for the genetic algorithm. For this purpose, we created random topologies, $G(n = 12$ to 22, $p = 0.25)$, again using the Erdős-Rényi model. Similar to the preceding evaluation scenario, we created a random set of target CPS containing $2n$ systems for each of the corresponding topologies. The results of executing the algorithms with different number of candidate solutions (average of 100 execution runs) is summarised in Figure 4. For this evaluation, we executed the greedy algorithm with 6 candidate solutions while the genetic algorithm was executed with 2, 4, and 6 candidate solutions. An obvious result of this experimentation was that the quality of solutions generated (Figure 4a) and the algorithm runtime (Figure 4b) increases with an increase in number of candidate solutions considered irrespective of the topology. Figure 4a further shows slight fluctuations in the quality of the solutions produced by the algorithms on various topologies. We attribute these fluctuations to the randomness of generating initial candidate solutions in both of these algorithms. Despite these fluctuations, we can infer that the genetic algorithms produce better quality results as compared to the greedy algorithm despite using far lower number of candidate solutions for all the topologies that were considered. For instance, the genetic algorithm considering 4 candidate solutions provided better solutions than the greedy algorithm considering 6 candidate solutions.

To summarise, the genetic algorithm can be fine-tuned using its input parameters like the number of candidate solutions it considers and the number of iterations it undergoes. The fine-tuning involves a trade-off between the quality of the solution it generates and the algorithm runtime.

VI. Discussion

Our control algorithms, which allocate edge-disjoint paths to the time-sensitive communication flows in manufacturing systems, perform well only with network topologies with high path diversity, e.g., in multi-rooted trees. Without high path diversity, it would be difficult to allocate edge-disjoint network paths to most time-sensitive flows. One might argue against allocating network links exclusively to communication flows. While data-center topologies do assign "non-conflicting paths" for some of their flows, they do so only for ensuring that bandwidth constraints on all network links are respected [19]. However, the QoS levels, in particular the communication latency and jitter, desired by time-sensitive flows of manufacturing system are a notch higher than the soft real-time communication flows in data-centers. Further, it has been shown that time-sensitive communication flows are negatively affected when network resources (like switch buffers, network links etc.) are shared across them [20]. In principle, allocating edge-disjoint network paths to these time-sensitive flows ensures that the desired QoS is provided to all these flows.

On the other hand, the time-sensitive flows constituting CPS, often, insufficiently load the network links assigned to it resulting in low network utilisation. We also agree that the approach of assigning network links exclusively for a communication flow is not scalable to systems consisting of thousands of time-sensitive flows. To address these issues, the time-sensitive flows must use the links in a time-multiplexed manner, i.e., the flows are assigned time-slots during which they have exclusive access to all the resources in a network path, but overall the network links are used by more than one flow. This will lead to an improvement in network utilisation and solve the scalability issue as well. The implementation of such an approach would require additional support not only from the underlying network but also from the end-hosts, for instance the ability to modify the queuing strategies in the network switches and accurate clock synchronisation among the end-hosts [5], [21]. However, we underscore the significance of the global view of the infrastructure provided through the logical centralization of software-defined environments which would simplify the control algorithms for such multiplexed access to network resources.

VII. Conclusion & Outlook

In this paper, we highlighted the importance of flexible ICT infrastructures for reconfigurable manufacturing system. To address the need for a flexible ICT infrastructure that provides desired QoS for manufacturing systems, we proposed an architecture of a software-defined environment with real-time properties. In particular, we focused on the networking domain by combining time-sensitive networking with software-defined networks. We also introduced a routing problem for time-sensitive communication flows in the manufacturing system along with a set of algorithms that efficiently solve this routing problem exploiting the logical centralization of the software-defined environments.

In our future work, we will address the issues discussed in Section VI and further strengthen this cornerstone for a "software-defined factory". We will also look into developing further concepts for the storage and compute domain to complement our algorithms on time-sensitive software-defined networking.

Acknowledgment

This research is supported by the German Research Foundation (DFG) as a part of projects within the *Graduate School of Excellence advanced Manufacturing Engineering* (GSaME) (www.gsame.uni-stuttgart.de) at University of Stuttgart.

References

[1] F. K. Pil and M. Holweg, "Linking product variety to order-fulfillment strategies," *Interfaces*, vol. 34, no. 5, pp. 394–403, 2004.

[2] G. Volpato and A. Stocchetti, "Managing product life cycle in the auto industry: evaluating carmakers effectiveness," *International Journal of Automotive Technology and Management*, vol. 8, no. 1, pp. 22–41, 2008.

[3] E. Westkämper, "Digital Manufacturing in the global Era," in *Digital Enterprise Technology*, pp. 3–14, Springer, 2007.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[5] M. Johas Teener, A. Fredette, C. Boiger, P. Klein, C. Gunther, D. Olsen, and K. Stanton, "Heterogeneous networks for audio and video: Using ieee 802.1 audio video bridging," *Proceedings of the IEEE*, vol. 101, pp. 2339–2354, Nov 2013.

[6] Z. M. Bi, S. Y. Lang, W. Shen, and L. Wang, "Reconfigurable manufacturing systems: the state of the art," *International Journal of Production Research*, vol. 46, no. 4, pp. 967–992, 2008.

[7] H. A. ElMaraghy, "Flexible and reconfigurable manufacturing systems paradigms," *International journal of flexible manufacturing systems*, vol. 17, no. 4, pp. 261–276, 2005.

[8] Y. Koren and M. Shpitalni, "Design of reconfigurable manufacturing systems," *Journal of manufacturing systems*, vol. 29, no. 4, pp. 130–141, 2010.

[9] P. Neumann, "Communication in industrial automation - What is going on?," *Control Engineering Practice*, vol. 15, no. 11, pp. 1332–1347, 2007.

[10] M. Felser, "Real-time ethernet–industry prospective," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1118–1129, 2005.

[11] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[12] F. Dürr and T. Kohler, "Comparing the Forwarding Latency of OpenFlow Hardware and Software Switches," Technical Report Computer Science 2014/04, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Parallel and Distributed Systems, Distributed Systems, July 2014.

[13] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren, "Practical TDMA for datacenter ethernet," in *Proceedings of the 7th ACM european conference on Computer Systems*, pp. 225–238, ACM, 2012.

[14] M. Blesa and C. Blum, "Ant colony optimization for the maximum edge-disjoint paths problem," in *Applications of Evolutionary Computing*, pp. 160–169, Springer, 2004.

[15] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[16] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, vol. 4, no. 2, pp. 65–85, 1994.

[17] P. Erdős and A. Rényi, "On random graphs I," *Publicationes Mathematicae 6*, pp. 290–297, 1959.

[18] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, (Pasadena, CA USA), pp. 11–15, Aug. 2008.

[19] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks.," in *NSDI*, 2010.

[20] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues Don't Matter When You Can JUMP Them!," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 1–14, USENIX Association, May 2015.

[21] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: a centralized zero-queue datacenter network," in *Proceedings of the 2014 ACM conference on SIGCOMM*, pp. 307–318, ACM, 2014.