Logistics Research (2020) 13:8
DOI_10.23773/2020_8

Decentralized Routing Algorithm with Physical Time Windows for Modular Conveyors

Simon Sohrt, Ludger Overmeyer

Received: 14 January 2020 / Accepted: 17 July 2020 / Published online: 25 August 2020

ABSTRACT

We describe a decentralized routing algorithm with physical time windows for modular conveying systems. Existing routing algorithms for modular conveyors are already capable of bi-directional conveying while avoiding conflicts such as collisions, deadlocks, livelocks and starvation effects. In addition to avoiding conflicts, routing algorithms must also select routes that minimize the transport time. No existing algorithm for modular conveyors bases this decision on the expected physical lead time, even though physical lead time directly affects the system throughput. In this publication, we present an algorithm that uses the physical lead time to select routes while avoiding conflicts. The avoidance of conflicts is mathematically proven and the algorithm's computational complexity is calculated. We present the system behavior of an exemplary layout which consists of nine modular conveying modules that are controlled by our algorithm. With only nine modules, the package throughput is on the same level as the package throughput of conventional sorting systems. Due to its modular design, additional modules can be added to further increase the throughput, thus surpassing the throughput of conventional sorting systems.

KEYWORDS: Modular conveyors · Multi-agent system · Decentralized control · Conflict-free · Bi-directional routing

Simon Sohrt
Corresponding Author
E-Mail: simon.sohrt@gmail.com

Ludger Overmeyer

Institute of Transport and Automation Technology,
Leibniz University Hannover, Germany

1. INTRODUCTION

Conventional material flow systems excel at achieving a high package throughput, but lack flexibility. Once installed in a warehouse, changing their configuration is expensive and impractical. Due to the increase of mass-customization and e-commerce, material flow systems must become more flexible to ensure quick response times to fast changing demands [1, 2, 3].

Modular conveying systems achieve high throughput while allowing for fast and flexible configuration changes. The layout can be changed within minutes or at most a few hours by adding more modules or by rearranging existing modules. Consequently, modular conveying systems can automate use-cases in warehouses that have been hard to automate with conventional systems. Automating these use-cases drives down costs while simultaneously increasing package throughput.

Furmans et al. [3] identified design patterns of modular conveying systems. The modules must have wheels underneath so that they can be quickly rearranged into a new layout. A decentralized control is needed to enable the modules to configure themselves by exchanging messages with their neighboring modules. No manual configuration of the system is necessary. In addition to being movable and having a decentralized control, all conveying systems must avoid conflicts to ensure the conveying of packages. The most critical conflicts are:

1. **Collisions** occur when two packages collide with each other and interlock.
2. **Deadlocks** occur when at least two occupied conveyors are cyclically waiting on each other to become available [4].
3. **Livelocks** occur when a package performs the same cycle of movements over and over again [5].
4. **Starvation** occurs when two packages from different directions compete at an intersection for right of way. If one direction has high

throughput and is always prioritized, the packages from the other direction will never get closer to their destination [6].

Conflict-avoidance is handled by routing algorithms. The need for routing algorithms is depicted in Fig. 1. In this example, four modular conveyors form an intersection. The intersection is surrounded by source and destination modules through which packages may enter and leave. In this example, the conveying of packages I, II, and III has already begun, while package IV just entered the system. The shortest path for every package is depicted by arrows. If package IV enters the intersection immediately and follows the shortest path, a conflict occurs (either a collision or a deadlock). Several routing algorithms have been developed that prevent conflicts as described in Section 2.

In addition to avoiding conflicts, routing algorithms must also select routes that positively affect the throughput. Existing routing algorithms have used different methods of selecting routes, but no algorithm has selected routes based on the expected travel time. In this paper, we propose a novel routing algorithm that selects routes based on the expected travel time by utilizing physical time windows. Since the throughput is a result of the travel time, the throughput is optimized by basing the selection of routes on the expected travel times.

Our proposed routing algorithm is designed for decentralized controlled modular conveyors. Every conveyor has a schedule and a clock which is synchronized through the network. The synchronized clocks are necessary to reserve physical time windows on the schedules. The conveyors reserve routes by exchanging messages: When a conveyor receives a request, it checks its schedule. If the request can be accepted, it sends a request to the next conveyor. If the request cannot be accepted, a new time is proposed to the requesting conveyor.

This paper is organized as follows. In Section 2, we present state-of-the-art routing algorithms for modular conveying systems and for automated guided vehicles (AGV). The modular conveyors are devided into large-scaled and small-scaled modules. We chose to include AGV routing algorithms because they can be modified to work in modular conveyors. Before describing the routing algorithm, we present our preliminary considerations regarding the synchronization of the conveyors' clocks in Section 3. In Section 4, we describe our algorithm for large-scaled modules. Based on the algorithm for large-scaled modules, we present a modified algorithm for small-scaled modules in Section 5. The characteristics of both algorithms are analyzed in Section 6. We give an application example in Section 7. The system behavior of a conveying system controlled by our algorithm is presented in Section 8. Finally, we present our conclusion and outlook in Section 9.

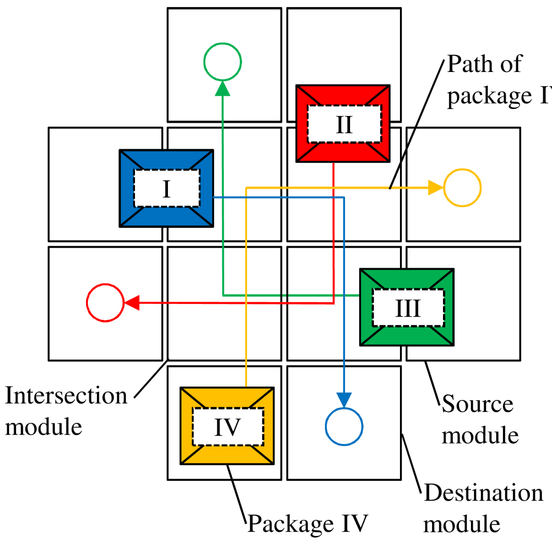

Fig.1: Intersection consisting of four modular conveyors (based on [6])

2. RELATED RESEARCH

In this Section we will first present the state-of-the-art for large-scaled modules, followed by small-scaled modules and at last automated guided vehicles.

We divide modular conveying systems into two classes based on their size in relation to the package size. If the largest package of a package spectrum is smaller than a single module, the modules are large-scaled in relation to the package size. Subsequently, if even the smallest package of a package spectrum is larger than a single module, the modules are small-scaled in relation to the package size. The majority of packages in a typical warehouse have a width between 100 mm and 600 mm and a length between 200 mm and 1000 mm [7]. Accordingly, in a typical warehouse, modules must be at least smaller than 100 mm × 200 mm to be considered small-scale.

We will only consider algorithms that can potentially fulfill our requirements, which are: conflict-free, work for any bi-directional conveyor layout, and select routes based on the expected travel time. We use a classification to filter out all algorithms that do not fulfill our requirements. Different classifications are available (see [8, 9, 10]), but in this publication, we use the classification by Seibold [6]. Only algorithms from Seibold's class of "Time-window based Route Reservation" are able to fulfill all our requirements. Algorithms from this class reserve routes, from source modules to destination modules, for individual packages. Every module keeps a schedule, which are used to reserve time windows for the individual packages. Even though we will not consider algorithms from other classes, we still want to mention the algorithms for modular conveying systems by Gue et al. [11, 2], by Mayer et al. [4, 12] and by Krühn et al. [13, 1].

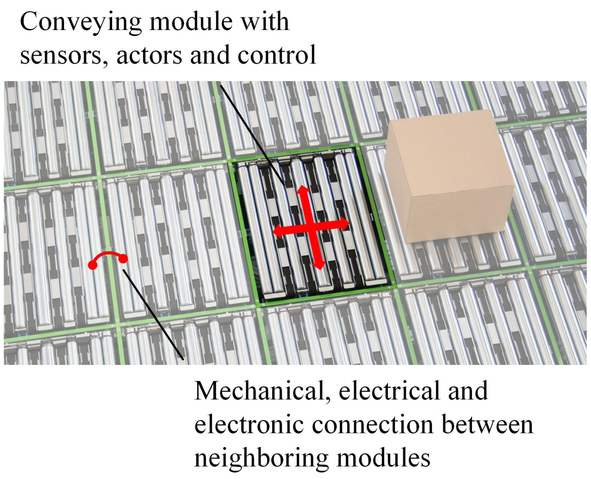
Conveying module with
sensors, actors and control
Mechanical, electrical and
electronic connection between
neighboring modules

Fig. 2: The GridSorter system is made up of several large-scaled conveying modules [6]

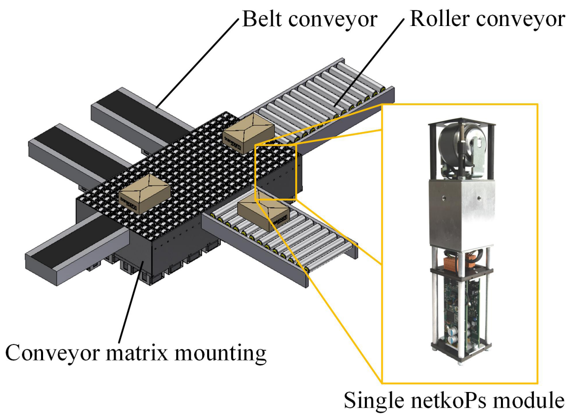

Fig.3: The netkoPs system is made up of several small-scaled conveying modules (based on [18])

2.1. Large-Scaled Modules

An example for a conveying system that is made up of large-scaled modules is shown in Fig. 2 – the GridSorter system. The prototypes of the GridSorter modules have been described in [4] and [6] and are now manufactured by *flexlog GmbH* [14]. The footprint of a GridSorter module is 500 mm × 500 mm allowing packages to be conveyed into all four cardinal directions. Every module has four sensors at the sides to detect incoming packages and a dedicated control. Communication is only possible with its four direct neighbors. All modules have wheels underneath and four standardized connections on every side, which are used to transmit information and power from one module to another. Due to the wheels and the standardized connections, the modules can be rearranged into a new layout within minutes. If multiple modules are combined to form a layout, no single control has an overview of the whole system. Instead, the modules exchange messages and decide the conveying of packages on their own. Thus, the modules are software-agents as defined by Franklin and Graesser [15] with decentralized and distributed controls. Large-scaled modules have also been manufactured by other companies. Two example systems are the XPlanar system [16] and the Motion Cube system [17], but no control algorithms have been published for these systems.

Seibold described a routing algorithm [6] that belongs to the class of “Time-window-based Route Reservation”. This algorithm is conflict-free and works for any bi-directional layout, selecting routes based on the expected logical lead time. One downside of utilizing logical time is that it does not progress unless a package is conveyed from one module to the next. The route with the shortest logical lead time is not necessarily the route with the shortest physical lead time.

2.2. Small-Scaled Modules

An example depicting a conveying system that is made up of small-scaled modules is shown in Fig. 3 – the netkoPs system. The prototypes of the netkoPs modules are described in greater detail in [18, 19, 20]. The netkoPs modules are based on the cogniLog modules that have been described in [1, 13, 7]. We have chosen the netkoPs modules as our example system because it is well documented by scientific publications.

Every netkoPs module has a sensor to detect packages, can convey in any direction, and has its own dedicated control. The footprint of a module is 60 mm × 60 mm. These modules can only communicate with their four direct neighbors and can be freely arranged into the slots of a matrix mounting. Prototypes of matrix mountings with different dimensions have been manufactured, and slots can be left unoccupied. Every matrix mounting has wheels underneath and can be freely positioned within a warehouse. Other examples of small-scaled conveyor systems include the Celluveyor from Uriarte et al. [21, 22, 23] and the “Magic Carpet System” presented by *Itoh Denki Ltd.* [24].

None of the published control algorithms for small-scaled modules belong to the class of “Time-windows based Route Reservation”, and therefore, are not presented. An important contribution was made by Krühn by introducing the concept of “module neighborhoods” for controlling small-scaled modules [13, 1]. By utilizing the “module neighborhoods” concept Krühn was able to modify an control algorithm originally designed for large-scaled modules to work on small-scaled modules.

2.3. Automated Guided Vehicles

Both the hardware and the algorithms for automated guided vehicles (AGV) are much more advanced than their modular counterparts. The following overview publications describe state-of-the-art AGV routing algorithms: [9, 10, 25, 8]. As with the modular conveying systems, we will only focus on algorithms from the class of "Time-window-based Route Reservation." Most algorithms from this class are based on the algorithm described by Kim and Tanchoco [26]. This algorithm is conflict-free, works for any bi-directional layout, and selects routes based on the physical travel time. Unfortunately, all routes are sequentially planned by a central control that has a complete overview of the system. After a route has been planned, it is uploaded to the respective AGV. Kim and Tanchoco even explicitly stated that the algorithm cannot be used in a decentralized controlled system: "Parallel execution of the routeing algorithm may result in conflicting travel schedules or gridlocks in a bidirectional network." [26]

We want to highlight two publications regarding Kim and Tanchoco's algorithm. The original algorithm neglected the problem of unexpected delays that can occur in real-world systems, during transport execution. Maza and Castagna [27] presented a modification that prevents conflicts even when the previously reserved time windows are missed. This is accomplished by monitoring intersections: If a package misses its time window on the intersection, no other AGV must enter the intersection until the delayed AGV has passed the intersection.

Möhring et al. [28] tested the practicality of the algorithm for real-world use-cases. Their algorithm is implemented to route vehicles at a Container Terminal in the Hamburg Harbor. Beforehand, they simulated different scenarios with up to 48 vehicles on a computer with an AMD-Athlon 2100+ (1,7 GHz) processor and 512 MB RAM. Even for worst-case scenarios (new routes must be calculated for all vehicles at the same time), the computation took less than half a second. We believe that these fast computation times are the reason why Kim and Tanchoco's algorithm was never modified extensively. It is already efficient enough for real-world use-cases even in its basic form.

Before we conclude the related research section, we want to mention the KARIS vehicles [29, 30]. The vehicles cannot only individually transport single items, but are also able to form two different functional clusters. As a discontinuous cluster, KARIS vehicles connect to each other to transport items that are larger than a single module. As a continuous cluster, several KARIS vehicles form a conveyor line to realize high throughput of goods. To our knowledge, no detailed routing algorithms for KARIS vehicles have been published.

To summarize, we identified two algorithms that are conflict-free and work in any bi-directional layout – Seibold's algorithm and Kim and Tanchoco's algorithm. Seibold's algorithm is decentralized, but routes are selected by using logical lead time and not physical lead time. Kim and Tanchoco's algorithm, and its modifications, use physical lead time to select routes but are designed for a centrally controlled system. As a result of our literature review, we identified the following **research question:**

Is it possible to create a decentralized routing algorithm that selects routes based on the physical lead time in any bidirectional layout while avoiding conflicts? We will tackle this question by utilizing physical time windows.

Algorithm 1: `ProcessMessages`

```
Input: Received messages
Result: Sent messages, modified schedule
delete routing entries with expired TTL;
if source module then
    StartNewRoute;
foreach unprocessed OBSOLETE message do
    delete affected entry from schedule;
    send OBSOLETE to succeeding module;
foreach unprocessed DENIAL message do
    modify affected entry according to DENIAL;
    delete affected entry from schedule;
    InsertIntoSchedule(modified entry);
foreach unprocessed REQUEST message do
    create new entry according to REQUEST;
    InsertIntoSchedule(new entry);
foreach unprocessed CONFIRMATION message do
    mark the affected entry as confirmed;
    if CONFIRMATION reaches source module then
        if iteration counter > maximum iteration then
            send RESERVE-UNLOCK with local package ID;
        set iteration counter to 0;
    else
        send CONFIRMATION to preceding module;
```

3 PRELIMINARY CONSIDERATION: CLOCK SYNCHRONIZATION

Since we want to reserve routes with physical time windows, we must first make sure that the clocks of all conveyors are synchronized. The clocks must not be synchronized perfectly, but the timing differences of neighboring conveyors must be small enough that the conveying process is not affected in a meaningful way.

The synchronization of clocks is a well-researched field [31], and many cheap technical solutions exist. We would like to point out the Precision Time Protocol (PTP), which was defined in the IEEE 1588-2008 standard [32]. PTP was designed for local systems requiring high accuracy that cannot bear the cost of a GPS receiver at each node, or for which GPS signals are inaccessible [33]. It can be used within a standard ethernet-network and is therefore cost-effective. The accuracy of PTP time signals is in the sub-microsecond range which is sufficient accurate for the conveying of packages which rarely exceeds 10 m/s.

Algorithm 2: `StartNewRoute`

```
Input: Received messages
Result: Sent messages, modified schedule
1  foreach unprocessed RESERVE-LOCK do
2      add to list of received locks;
3  foreach unprocessed RESERVE-UNLOCK do
4      remove from list of received locks;
5  if local package without routing entry then
6      if priority of local package higher than priority of received locks then
7          if iteration counter $>$ maximum iteration then
8              send RESERVE-LOCK with package ID and priority;
9          create routing entry for local package;
10         calculate TTL for newly created routing entry;
11         InsertIntoSchedule(newly created entry);
```

Algorithm 3: `InsertIntoSchedule`

```
Input: Routing entry
Result: Sent messages, modified schedule
1  filter out all requested entries with a lower priority;
2  insert routing entry into filtered schedule;
3  if insertion successful then
4      if REQUEST reached destination then
5          send CONFIRMATION to predecessor;
6      else
7          send REQUEST to neighbor with shortest lead time;
8      foreach overwritten entry do
9          send DENIAL to the preceding module;
10         send OBSOLETE to the succeeding module;
11 else
12     send DENIAL to preceding module;
```

4 ALGORITHM FOR LARGE-SCALED MODULES

As previously mentioned in Section 3, we assume that the clocks of every module are sufficiently synchronized. The main algorithm is described in pseudocode in Alg. 1, and its two subroutines are described in Alg. 2 and Alg. 3. For further clarification an application example is given in Section 7.

4.1 Overview of the Algorithm

A short overview of the algorithm: The modules reserve a route from source to destination by exchanging messages. Every module along a route stores a routing entry on its schedule. Routing entries include an arrival and a departure time. Every route reservation has two phases: the request phase and the confirmation phase. In the request phase the route is planned from source to destination. During this phase, the modules have to select a path, and they have to negotiate the arrival and the departure times. After the request has reached the destination module, the route is confirmed from destination to source.

The presented algorithm ignores the effects of acceleration, deceleration, and slippage that occur in real-world conveying systems. All three effects can be accounted for by adding a safety allowance to the time windows. In the following paragraphs, we will not explicitly mention this safety allowance for the sake of simplicity.

Before route reservations can be created, the modules must be initialized. During this initialization, every module generates a routing table by communicating with its direct neighbors. Hereupon, this information is propagated through the system. This process is similar to creating metric tables in computer networks using the distance-vector routing protocol [34]. For every destination, the modules determine the ID of the neighboring module that is closest to this destination, the distance to the destination, and the number of intersections between the source module and the destination.

Routing entries store the following data: unique request ID, physical time window (arrival time and departure time), priority, time-to-live (TTL), ID of the preceding module, ID of the succeeding module, and state (the two states are "requested" and "confirmed"). The unique request ID, the priority and the TTL are set by the source module at the creation of the request and are not altered by the other modules along the route.

Modern warehouses have already assigned a unique ID to every package, which the source modules reuse. If an external process does not set a priority, the source module generates a priority instead. This generated priority is based on the timestamp at which the package was first introduced to the source module. Routing entries with earlier timestamps have a higher priority than entries with later timestamps. Basing the priority on the introductory time has the advantage that starvation effects are avoided. We describe the avoidance of starvation effects in more detail in Section 4.2.

The TTL is the maximum time a routing request may "live" before it is discarded and a new request is created by the source. Consequently, every module periodically checks the TTL for routing entries and deletes expired ones. The introduction of TTL was necessary to take into account the uncertainty of a decentralized controlled system with physical time windows. Since no module has a complete overview of the system, the source module can only estimate how long the route reservation process takes. We will present how the TTL is estimated in Section 4.3, after we overview the algorithm.

Modules can send the following six message types which must be processed in the following sequence: OBSOLETE, DENIAL, REQUEST, CONFIRMATION, RESERVE-LOCK, RESERVE-UNLOCK. The processing of messages in this sequence positively affects the throughput. Both OBSOLETE and DENIAL messages delete routing entries on the schedule. By processing them first, the schedule opens up. Then, the REQUEST and CONFIRMATION messages are processed, which either create or change the state of routing entries. By processing the REQUEST and CONFIRMATION messages last, previously unavailable time windows can be reserved and the chance of reserving a well-fitting time window is increased, positively affecting the throughput. The LOCK-RESERVE and the UNLOCK-RESERVE messages are only sent and processed by the source modules. Their meaning will be explained when describing the initial creation of a REQUEST message at a source module in the next Section.

4.2 Initial Creation of a REQUEST Message at a Source

When a package enters the conveying system on a source module, the schedule of this source module becomes blocked indefinitely until the route reservation is confirmed. Due to the decentralized nature of the system, the source modules are not synchronized. Caused by this lack of synchronization, it is possible, yet very unlikely, that a source module may never finish a route reservation and starves. An example: The module m_n sends a REQUEST to its optimal neighbor m_{n+1}. Since m_{n+1} has already a confirmed routing entry at the requested time, a DENIAL is sent back to m_n. m_n must now alter the time window of the corresponding entry. This altered time window now overlaps with a confirmed routing entry. Consequently, m_n must send a DENIAL back to its preceding module m_{n-1}. This chain of events may repeat itself, until the TTL expires. Since there is no mechanism for avoiding the same chain of events in the next iteration attempt, the source module starves. To prevent this starvation, a synchronization lock is introduced: Every source module keeps track of the number of failed reservation attempts. If a user-specified maximum number of iteration attempts is exceeded, the source module sends a RESERVE-LOCK to all other source modules. This RESERVE-LOCK includes the package ID and the priority of the package. All source modules keep a list of the received RESERVE-LOCKS. Source modules only start new route reservations, if the local package has a higher priority than all of their received RESERVE-LOCKS. The algorithm for starting a new route is shown in Alg. 2.

When a matching CONFIRMATION message reaches a source module, the iteration counter for the number of failed attempts is reset back to 0. If the source module has previously sent a RESERVE-LOCK, a RESERVE-UNLOCK is sent.

Due to the introduction of synchronization locks, the conveying system may temporally become centrally controlled, because every source module may have to wait on a single source module to finish its reservation process. Since every source module has to wait, the throughput of the conveying system is negatively affected. However, the circumstances under which a synchronization lock are used are very unlikely to occur and therefore the throughput will be not affected at all in most conveying systems. In Section 8 we analyze the system behavior of a simulated conveying system controlled by our algorithm. No RESERVE-LOCK message was sent.

If the source module is not locked, it estimates the TTL and the desired arrival time, and subsequently, sends a REQUEST message to the neighboring module with the shortest lead time. Every REQUEST message includes a unique request ID, the priority, the destination ID, the desired arrival time, and the TTL.

This desired arrival time of the REQUEST must be far enough into the future that the route reservation process is most likely completed by then. The completion time depends on the number of modules between the source and the destination and the number of overlaps that will likely occur, which need to be resolved. Since the source has no complete system overview, it can only estimate the number of overlaps. Because the number of overlaps can only be estimated, the arrival time can only be estimated.

If more overlaps occur than estimated, the CONFIRMATION message will reach the source after the estimated arrival time and it would become physically impossible for the package to arrive at the estimated arrival time. Because the first time window is missed, all subsequent time windows may also be missed. A mismatch would have occurred between the projected movement of the package and the actual movement. As already mentioned in Section 2.3 Maza and Castagna [27] have proven that even when a mismatch occurs, conflicts can still be avoided by following the sequence of the previously negotiated time windows.

Even though no conflict occurs due to the mismatch, it is still advantageous to avoid such a mismatch because the selection of routes depends on accurate schedules. The selection of routes is based on the lead time, which itself, is based on the projected movement. If the mismatch becomes too big, sub-optimal routes might be chosen.

To avoid a mismatch between projected movement and actual movement, we introduce the concept of TTL. Before a source sends the first REQUEST message, it estimates how long the route reservation will most likely take. A safety margin is added to increase the chance of completing the route reservation within the TTL. If the TTL is estimated too conservatively and the route reservation is confirmed faster than estimated, the package *can not* start earlier, as this would also cause a mismatch between projected movement and actual movement. Therefore, the package has to wait and the throughput is negatively affected. If the estimate is too optimistic and the route reservation is not confirmed within the TTL, the source module has to start a new iteration attempt. Since the package has to wait until this new iteration attempt is completed, the throughput is once again negatively affected. In the next paragraphs, we present our approaches for estimating the TTL.

4.3 Estimating the TTL

We developed two methods for estimating the TTL. The first method bases the estimate mainly on the average time previous route reservations took before they were confirmed and is depicted in Eq. 1. Consequently, every source must store the average time of all previously confirmed route reservations for every destination. The average time of previous route reservations is the first term in the equation and is written as $\bar{t}_{dest}$. Since route reservations which were not confirmed within the TTL, increase the value of $\bar{t}_{dest}$, this estimation method is self-correcting.

The second term $n_{safe} \cdot i$ takes into account that the number of packages within the system can randomly peak. This package peak increases the chance of routes conflicting with each other. Due to conflicting routes, route reservations take longer than average, since conflict avoidance requires the sending of additional messages. To avoid a source from starving, every source increases an iteration counter i by 1 for every failed attempt. This iteration counter starts at 1 for every newly started request and is reset back to 1 once the reservation has been successfully completed. i is then multiplied with the safety factor n_{safe}. While testing on different layouts, we achieved satisfactory throughputs by picking a value in the range from 1 to 2 for n_{safe}. In Section 8.2 we present the resulting conveying times for $n_{safe} = 1$ for an exemplary layout. The formula in its entirety is as follows:

$$TTL_m(\bar{t}_{dest}, i) = \bar{t}_{dest} + n_{safe} \cdot i \tag{1}$$

If the layout of the conveying system changes, all previously calculated averages become invalid. Since packages still need to be conveyed during this start-up phase, a second method for estimating the TTL is needed:

$$TTL_s(n_{dest}, i) = 2 \cdot t_{com} \cdot n_{dest} \cdot i \tag{2}$$

where n_{dest} is the number of modules between source and destination and t_{com} is the time for one module to receive and process a message from a neighboring module. The factor 2 is needed because all modules along a route have to receive and process at least two messages – the REQUEST message and subsequently the CONFIRMATION message. i is once again the iteration counter that starts at 1 and is increased by 1 until the route reservation has been successfully confirmed. After completing the route reservation, i is resetted. The system switches from TTL_s to TTL_m as soon as $\bar{t}_{dest}$ can be computed.

4.4 Selecting a route

Routes are selected based on the physical lead time. The physical lead time is the sum of the base lead time and temporary increases caused by previously received DENIAL messages. The base lead time for every neighboring module is computed by dividing the destination's distance (which was determined during the initialization phase) by the uniform conveying speed of the modules. Every DENIAL message includes a proposed time, when the sending module is ready to accept a package. This proposed time causes a temporary increase of the lead time. Modules always send REQUESTS to the neighboring module with the lowest lead time. Due to temporary lead time increases, the REQUESTS are not necessarily sent along the shortest path. Livelocks are avoided by never choosing the preceding module, even if it has the lowest lead time.

4.5 Processing Messages

When a module receives a REQUEST message, it creates a routing entry with the requested arrival time. Next, all previously created requests with a lower priority are filtered out from the schedule. The module then attempts to add the newly created entry to the filtered schedule without creating overlaps. Testing for overlaps is done by using the algorithm described in [35]. If the newly created entry cannot be inserted into the schedule, a DENIAL is sent back to the preceding module that includes the next available arrival time. If the newly created entry is successfully inserted into the schedule and reached its destination, a CONFIRMATION is sent back to the preceding module. If the newly created entry is successfully inserted, but did not reach its destination, a REQUEST is sent to the neighboring module with the shortest lead time. Livelocks are prevented by excluding the possibility of sending a REQUEST message to the predecessor.

For every entry that has been overwritten by the newly created routing entry, a DENIAL message is sent to its preceding module and an OBSOLETE message is sent to its succeeding module. The algorithm for inserting into the schedule is shown in Alg. 3.

When a module receives a CONFIRMATION message, it changes the state of the corresponding routing entry to the state confirmed and sends a CONFIRMATION message to its preceding module. When a CONFIRMATION message reaches the source module, the actual conveying of the package can start. The source module also sends a RESERVE-UNLOCK, if it has previously sent a RESERVE-LOCK. The iteration counter for the number that tracks the number of failed attempts is reset back to 0.

A DENIAL message includes the unique request ID and the proposed time. When a DENIAL message is received, the receiving module first deletes the corresponding entry from its schedule. A new entry is created with a modified departure time. This modified entry is then added to the schedule using the algorithm described in Alg. 3.

5 MODIFIED ALGORITHM FOR SMALL-SCALED MODULES

The algorithm is modified for small-scaled modules by introducing the concept of module neighborhoods described by Krühn in [13]. Every message must be extended to include the package dimensions, which is used to determine the size of the neighborhood. Neighborhoods are made up of one master module and several slave modules. Routing entries must be extended by including the ID of the module which created the entry (the module itself or a remote master module). Furthermore, three more messages must be introduced: SCHEDULE-REQUEST, SCHEDULE, and SCHEDULE-ACKNOWLEDGE. To keep this paper concise, we will only demonstrate the usage of these three new messages when a master module must process a REQUEST message.

If a small-scaled module receives a REQUEST message from a preceding master module, it becomes the master module for a new neighborhood. The master module now sends a SCHEDULE-REQUEST message to all slave modules. The SCHEDULE-REQUEST message consists of the ID of the master module, the priority of the REQUEST message, and the package ID. All modules keep a list of all received SCHEDULE-REQUEST messages. The modules always process the SCHEDULE-REQUEST message with the highest priority first. Processing is done by sending their own schedule to the master module via a SCHEDULE message.

After receiving the schedules of all modules within the neighborhood, the master module attempts to insert the newly created routing entry into all schedules. If the insertion fails on only one schedule, the master module sends the unmodified schedules back to the slave modules via SCHEDULE messages and sends a DENIAL message to the preceding master module. If the insertion is successful, the master module sends back the modified schedules to all slave modules. When the slave modules receive the modified schedule they first check, if their schedules have been altered in the meantime by either themselves or another master module. If this is the case, they re-send their schedule via a SCHEDULE message to the master module. The master module must now try once again to insert the newly created routing entry into all schedules. If the schedules have not been changed, the slave modules send back a SCHEDULE-ACKNOWLEDGE message to the master module, indicating that they have accepted the modification of their schedule. After the SCHEDULE-ACKNOWLEDGE message has been sent, the corresponding SCHEDULE-REQUEST message is removed from the SCHEDULE-REQUEST list of the slave module.

After receiving the SCHEDULE-ACKNOWLEDGE message, the master module sends a REQUEST to the optimal neighbor. A successful insertion is depicted in Fig. 4 for a package that has the dimensions of 3 × 3 modules. The reserved neighborhoods are shown for t = 10 ms and t = 11 ms.

When a routing entry is deleted on a slave module by a REQUEST with a higher priority, the slave modules informs the master module by sending a SCHEDULE message. The master module then sends a DENIAL message to all other slave modules in the neighborhood and an OBSOLETE message to its succeeding master module and a DENIAL message to its preceding module.

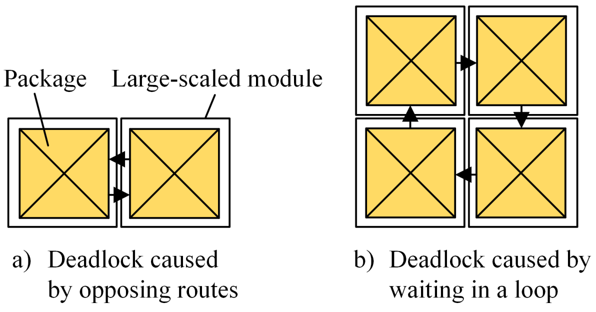
Package
Large-scaled module
a) Deadlock caused by opposing routes
b) Deadlock caused by waiting in a loop

Fig.4: Neighborhoods during route reservation (based on [13])

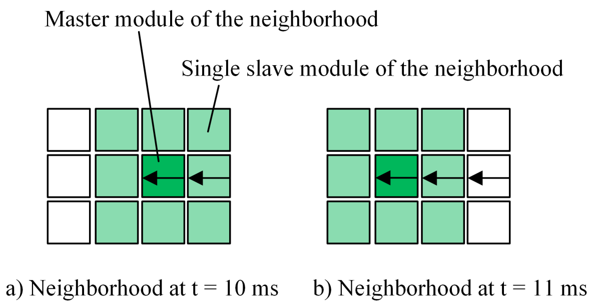
Master module of the neighborhood
Single slave module of the neighborhood
a) Neighborhood at t = 10 ms
b) Neighborhood at t = 11 ms

Fig.5: Deadlock situations that must not occur [6]

6 CHARACTERISTICS OF THE ALGORITHMS

In this Section we prove the absence of conflicts and calculate the computational complexity of the algorithms.

6.1 Proof of Conflict Absence

Conflicts include collisions, livelocks, starvation effects and deadlocks. We avoid livelocks by omitting the requesting module, when searching for a new optimal module. Starvation at sources is avoided by using REVERSE-LOCK messages. In the following paragraphs, we will prove the absence of collisions and deadlocks.

Seibold identified two possible deadlock situations that can occur in large-scaled modular conveying systems [6]: deadlocks caused by opposing routes and deadlocks caused by packages waiting in a loop (see Fig. 5). In both cases, the packages are deadlocked because the time window on the succeeding module overlaps with the time window of another package. Accordingly, deadlocks and collisions are avoided by guaranteeing that no time windows overlap. Since the arrival and departure times of packages on every module are solely determined by either accepting or denying REQUEST messages, we must only examine this phase. The following proof works for both small-scaled and large-scaled modules. In the case of large-scaled modules, only the schedule of the neighboring module must be checked for overlaps. In the case of small-scaled modules, an additional step is needed: When a module receives a REQUEST message, it must check the schedules of all modules within the neighborhood before either a REQUEST or a DENIAL can be sent (see Section 5). To keep this publication concise, we omit this additional step in the following paragraphs. We first prove the absence of overlapping time windows for the case of opposing routes by proof of exhaustion. The following cases can occur:

1. Both entries are in the state requested: Both modules send REQUEST messages to the other module. The requested time window will overlap on both modules with the already existing routing entries. Therefore, both modules will compare the priority of the entry with the priority of the REQUEST message and delete the one with the lower priority. As a result, both modules will end up with only the entry for the package with the higher priority and no deadlock occurs.

2. One entry is in the state confirmed, the other is in the state requested: Only the module with the requested entry sends a message. The module with the already confirmed entry receives the REQUEST, but immediately sends back a DENIAL, because already confirmed requests must not be deleted by a REQUEST message (even if it has a higher priority) and no deadlock occurs.

3. Both entries are in the state confirmed: This case cannot occur and therefore no deadlock occurs. Prior to having confirmed entries, every entry has to be in the requested state. If both neighboring modules were in the state requested at the same time, the rules for case 1 avoid the creation of two opposing entries with the state confirmed. If one of the entries was in the state confirmed and the other was in the state requested, the rules from case 2 avoid the creation of two opposing entries with the state confirmed.

In contrast to deadlocks caused by opposing routes, deadlocks in loops are caused by non-opposing routes. Once again, we will prove the absence of overlapping time windows by proof of exhaustion. In the following, we examine a module in a loop that receives a request from a module outside the loop. The module within the loop can be in two different states:

1. The entry of the receiving module is in the state confirmed: The receiving module rejects the REQUEST and sends back a DENIAL with a new proposed time and no deadlock occurs.

2. The entry of the receiving module is in the state requested: The module that receives the request compares the priorities of the REQUEST message with the priority of the routing entry, deleting those with lower priority and no deadlock occurs.

6.2 Computational Complexity

We first determine the computational complexity of the algorithm for large-scaled modules before moving on to the modified algorithm for small-scaled modules. The computational complexity of the algorithm for large-scaled modules depends on the maximum number of messages a module receives. The processing time of OBSOLETE, CONFIRMATION, RESERVE-LOCK, and RESERVE-UNLOCK messages is constant. On the contrary to this, the processing time for REQUEST or DENIAL messages is not constant, since the modules have to check for overlaps when inserting a newly created routing entry into their schedules. A single module can at most have as many routing entries as there are packages in the system, because we do not allow backtracking and consequently no package can have more than one corresponding routing for a package. In the worst-case, a module has to check for overlaps with every routing entry on its schedule. Therefore, the computational complexity for large-scaled modules becomes $\mathcal{O}(m_o + m_{rd} \cdot p)$, where m_o is the number of received OBSOLETE, CONFIRMATION, REVERSE-LOCK, and REVERSE-UNLOCK messages, m_{rd} is the number of received REQUEST and DENIAL messages, and p is the number of packages in the system.

The computational complexity for small-scaled modules is higher, since additional messages need to be sent and processed to coordinate all modules within the neighborhood. The processing of the SCHEDULE-REQUEST, SCHEDULE, and SCHEDULE-ACKNOWLEDGE messages is constant. Processing of OBSOLETE, CONFIRMATION, RESERVE-LOCK, and RESERVE-UNLOCK messages is no longer constant, since the master module needs to forward these messages to all modules within the neighborhood. In order to process REQUEST and DENIAL messages, the master module must both check its own schedule and the schedules of all slave modules. Therefore, the computational complexity for small-scaled modules is $\mathcal{O}(m_s + n \cdot (m_o + m_{rd} \cdot p))$, where m_s is the number of received SCHEDULE-REQUEST, SCHEDULE, and SCHEDULE-ACKNOWLEDGE messages by other master modules, n is the number of modules within the neighborhood, m_o is the number of received OBSOLETE, CONFIRMATION, RESERVEE-LOCK, and RESERVE-UNLOCK messages, m_{rd} is the number

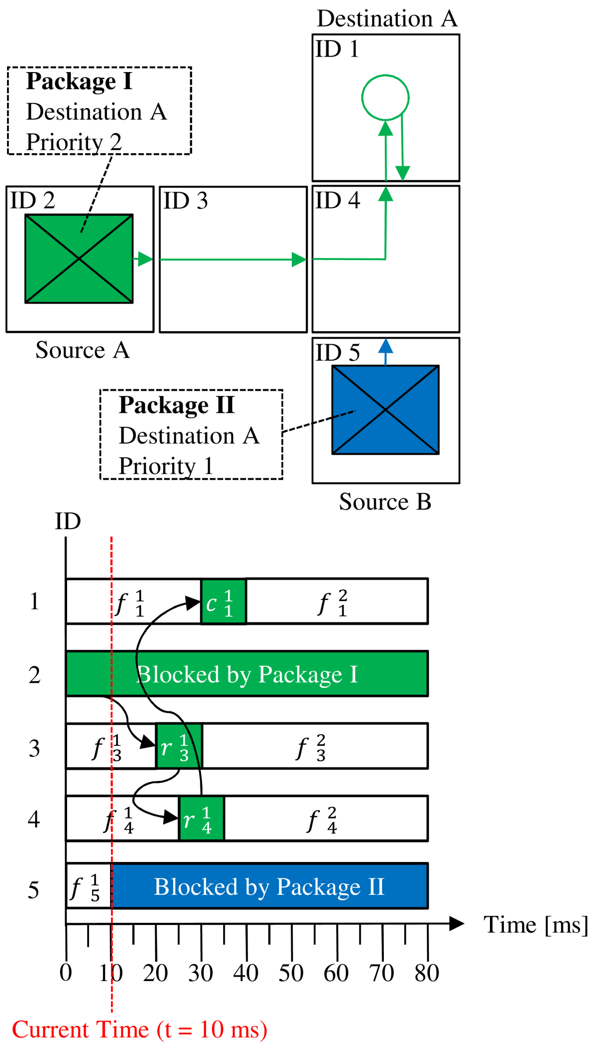

Fig.6: Application example at t= 10 ms

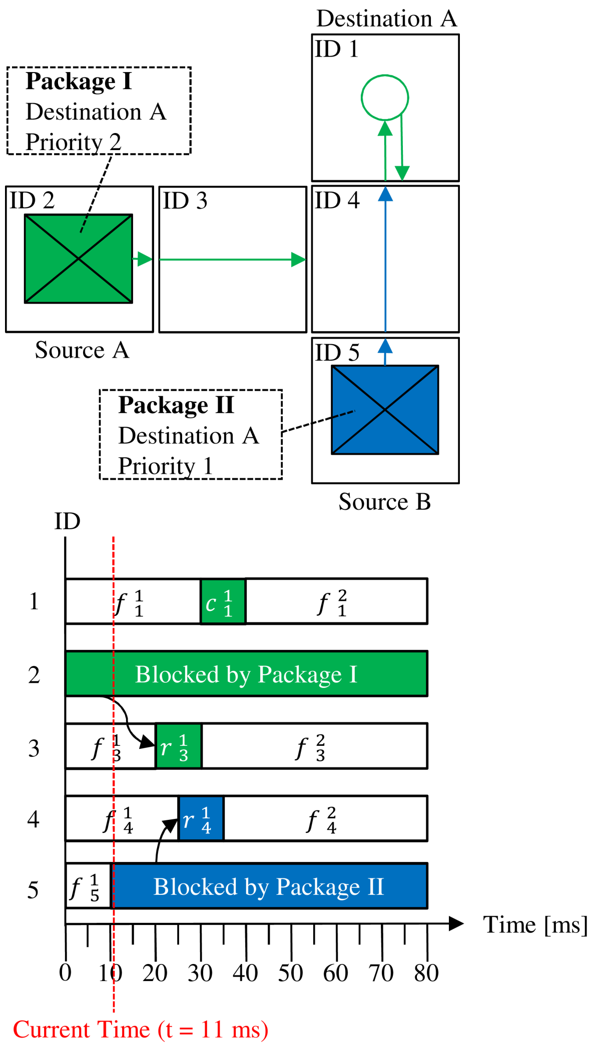

Fig.7: Application example at t= 11 ms

of received REQUEST and DENIAL messages, and *p* is the number of packages in the system.

7 APPLICATION EXAMPLE

In the Fig. 6 to 10, the system behavior is shown for large-scaled modules. In every figure, the layout is shown on the top, and the schedules of the modules are shown on the bottom. No module has this complete overview of the system – every module can only access its own schedule.

The layout consists of five modules. The unique identification number (ID) of every module is in their top left corner. Two source modules exist in this layout: source A has the ID 2 and is at the left-most of the layout, and source B has the ID 5 and is at the bottom-most of the layout. There is only one destination module in the layout: It has the ID 1 and is at the top-most of the layout. Routing entries that are in the state requested are displayed as a single arrow on a conveyor. Routing entries in the state confirmed are displayed with two arrows on the conveyor. Routing entries are assigned to their packages in two ways: The first arrow originates at their respective package, and the arrows have the same color as the packages.

The schedules of all modules are depicted on the right side: The horizontal axis displays the time, while the schedules of the individual modules are arranged on the vertical axis. The states of the time windows are denoted by one of three possible letters: *f* meaning the time window is not assigned to any routing entry, *r* meaning the time window is assigned to a routing entry that is in the state requested and *c* meaning the time windows is assigned to a routing entry that is in the state confirmed. Every time window has two indexes: The lower index is used to denote the module ID, and the upper index is used to number the time windows.

In this application example, receiving and processing a message from a neighboring module takes 1 ms. Conveying a package from the center of a module to a neighboring module takes a minimum of 10 ms. We

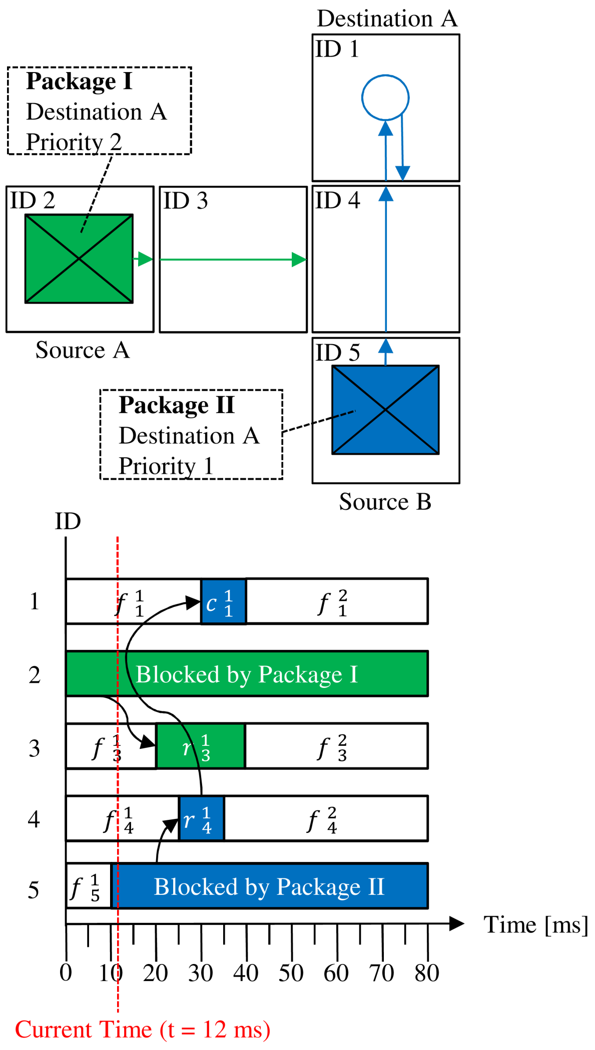

Fig.8: Application example at t= 12 ms

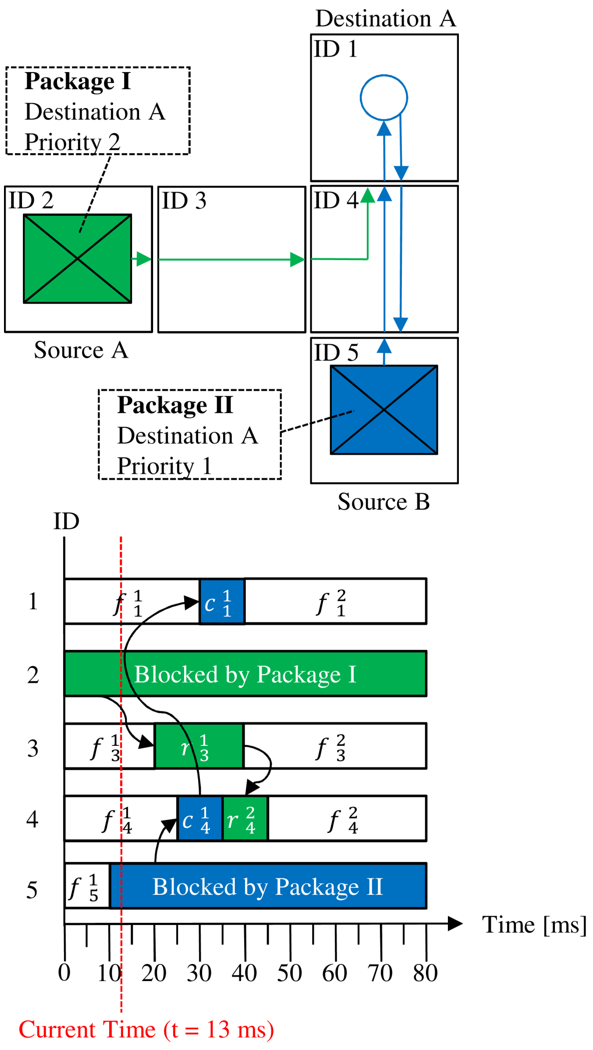

Fig.9: Application example at t= 13 ms

have chosen this unrealistically high conveying speed in this example for the following reason: Since the conveying speed and the communication speed are now within the same magnitude, it becomes possible to display both in the same figures.

At t = 10 ms, package I has entered the system through module 2 (source A), and its request has reached module 1 (destination A), which sends back a CONFIRMATION message to module 4. At the same time, package II enters the system through module 5 (source B). Module 5 sends a REQUEST message for package II to module 4. Package II has a higher priority than package I.

At t = 11 ms, module 4 receives both the CONFIRMATION message for package I from module 1 and the REQUEST message for package II from module 5. As stated in Section 4, REQUEST messages are always processed before CONFIRMATION messages. Therefore module 4 processes the REQUEST message for package II first: Since the requested arrival times for package I and package II overlap, the module compares the priorities of the packages, and sub sequently, deletes the routing entry for package I because its priority is lower than the priority of package II. Module 4 then creates a new routing entry for package II and sends the following three messages. Firstly, an OBSOLETE message is sent to module 1 to inform it that the already confirmed routing entry for package I must be deleted. Secondly, a DENIAL message is sent to module 3 with a new proposed arrival time for package I. Finally, a new REQUEST message for package II is sent to module 1. After module 4 has finished processing the REQUEST message, it processes the CONFIRMATION message. Since the routing entry the CONFIRMATION message is referring to has been deleted, the message is discarded.

At t = 12 ms, module 1 receives two messages from module 4: the OBSOLETE message for package I and the REQUEST message for package II. The reception of the OBSOLETE message causes the deletion of the previously confirmed time window for package I.

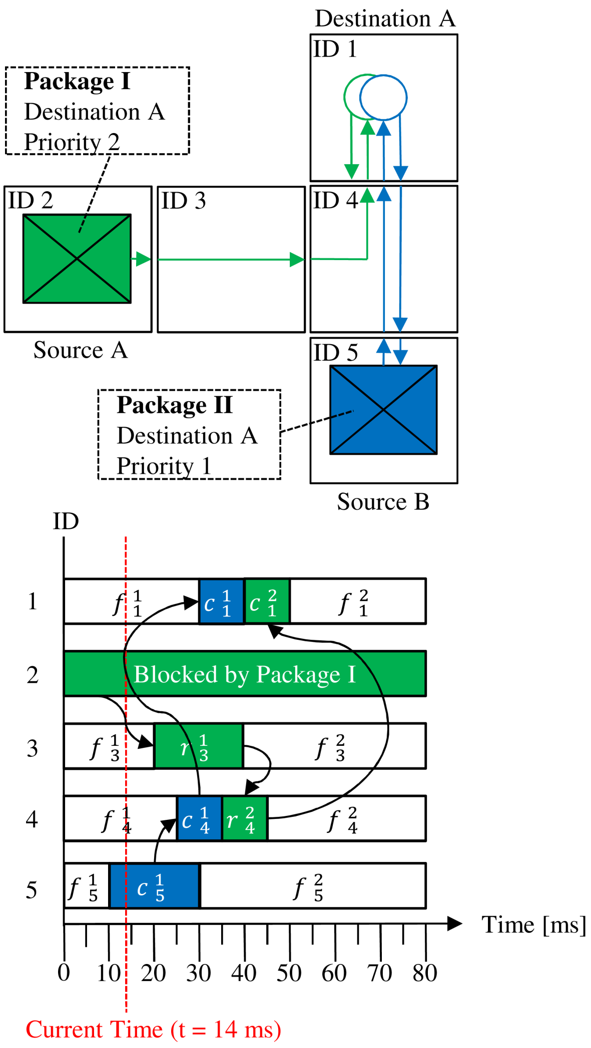

Fig. 10: Application example at t= 14 ms

Caused by the reception of the REQUEST message for package II, module 1 creates a routing entry for package II. This routing entry is in the state confirmed because the request has reached its destination. Subsequently, module 1 sends a CONFIRMATION message back to module 4. At the same time, module 3 receives the DENIAL message from module 4 and extends the time window for the routing entry for package I. Module 3 sends a new REQUEST message with an altered arrival time for package I to module 4.

At t= 13 ms, module 4 receives two messages: the CONFIRMATION message from module 1 for package II and a new REQUEST message from module 3 for package I. Since the requested arrival time for package I does not overlap with the time window of package II, a routing entry for package I is created on module 4. Subsequently, module 4 sends a REQUEST message for package I to module 1. Next, the CONFIRMATON message from module 1 is processed: The state for the routing entry on module 4 for package II is set to confirmed, and module 4 sends a CONFIRMATION message to module 5.

At t = 14 ms, module 1 receives a REQUEST message for packageI from module 4: It checks its schedule and creates a routing entry for package I. This routing entry is in the state confirmed, because the request has reached its destination. Subsequently, module 1 sends a CONFIRMATION message for package I back to module 4. At the same time, module 5 receives the CONFIRMATION message for package II. Module 5 alters the state of the corresponding routing entry to confirmed and the reservation process for package II has successfully ended. The schedule of module 5 is no longer blocked indefinitely by package II.

To show the effects of a sub-optimal calculation of the TTL, the TTL is calculated conservatively in this example. Every source module calculates the TTL by adding 10 ms to the creation time of a request. Because of this sub-optimal TTL calculation, package II has to wait for 6 more ms after the CONFIRMATION message has reached source module 2 before the conveying can start.

8 SYSTEM BEHAVIOR

In this Section we present our simulation that we have used to both validate the routing algorithm and measure its performance. We will first show the set-up of our simulation, followed by an analysis of the system behavior of a typical layout.

8.1 Simulation Set-Up

Our simulation is built on top of the MASON simulation library core version 17 [36]. Since our simulation set-up is similar to the simulation set-up described in [6], we will only present the most important simplifications. The first simplifications are that the clocks of the conveyors are perfectly synchronized at all times and that no hardware or communication malfunctions occur. Just like their real-life counterparts, the simulated modules can only communicate with their four neighboring modules.

The simulated modules have the same base area as the GridSorter modules – 500 mm × 500 mm. The simulated packages have a base area of 480 mm × 480 mm and consequently one module can carry at most one package. Inertia effects and slippage are not modeled. Every source module has a packages queue. During the simulation, packages are added to this queue faster than they are removed. The destinations of the packages are randomly chosen. As soon as a source module becomes unoccupied, the longest waiting package from the queue is placed on the source. Source modules only start the reservation process, when the package is placed on them. Destination modules remove packages as soon as the package center arrives at the center of the destination module.

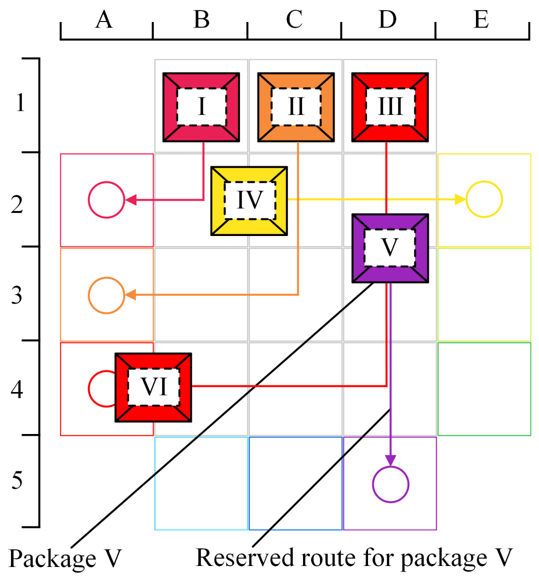

Fig. 11: Simulated exemplary layout

Our example layout is depicted in Fig. 11 and consists of 9 regular modules, 3 source modules, and 9 destination modules. Every destination module is distinguished by its own unique color. Packages are colored in the same color as their destination. The source modules are at the top (row 1), and the destination modules are at the sides (column A and E) and at the bottom (row 5). A video of a real-world system with a similar layout (but controlled by a different routing algorithm) is shown in [37].

We define one simulation step as equal to 1 ms in the real-world. Accordingly, our resolution for measuring times is ±0.5 ms. The simulated communication speed and the simulated conveying speed can be freely defined. We set the time for sending and receiving a message from a neighboring module to 1 ms. The conveying speed is set to 2 m/s. Since the modules are 500 mm × 500 mm, conveying a package from one module to another takes 250 ms. We performed 100 simulation runs – each one with a different seed value for our random number generator. The random number generator determines the destination for every package. We simulated 3 600 000 steps per simulation run, which is equivalent to one hour in real-world time.

During the simulation we have set the maximum number of iteration attempts i_{max} to 10. No RESERVE-LOCKS were sent during all simulation runs. Subsequently, this aspect of the algorithm will not be analyzed in the next Section. The value of n_{safe} (see Eq. 1) was set to 1.

8.2 Statistical System Behavior Analysis

We calculated the sample mean and the standard error of mean (SE) for all measurements. All values that we measured were normally distributed. The sample mean of the conveyed packages over all runs is 8 598 and its SE is 6.7. Conventional sorting and distributions systems have a throughput between 5 000 and 10 000 packages per hour according to [38]. A modular conveying system with only 9 modules controlled by our algorithm already performs as good as a conventional system. Due to its design, additional modules can be added to further increase the throughput and the throughput of conventional systems is surpassed.

We measured the mean transport time and its SE for every combination of source and destination. The transport time is the sum of the route reservation time and the conveying time. The results are depicted in Table 1. We also included the minimal achievable time for every combination of source and destination to make the assessment of the performance of the algorithm easier.

We explain the calculation of the minimal transport time by closer examining the value for source B1 to destination A2, which is in the left-most and top-most green highlighted value in the Table. Before the conveying begins, the route must be reserved. The shortest path for the REQUEST is B1→B2→A2. The shortest path for the CONFIRMATION is A2→B2→B1. Since sending and receiving a message takes 1 ms, the minimal achievable route reservation time is 4 ms. The shortest path for the conveying time is identical to the shortest path of the REQUEST message. Since conveying a package from one module to a neighboring module takes

250 ms, the minimal achievable conveying time is 500 ms and the minimal transport time from source B1 to destination A2 is 504 ms.

Minimal transport times can only be achieved by our algorithm, if the shortest paths of all packages that are concurrently in the system do not intersect. This is illustrated in Fig. 11: The conveying of package IV to destination E2 has already begun. Package I, II, and III cannot start, even though their routes have been confirmed, because first package IV has to pass through module B2, C2, and D2 and their transport times are negatively affected.

By dividing the mean transport time by the minimal transport time, we can assess which routes differ the most from the minimal transport times. The most negatively affected routes are the routes to the destinations A2 and E2 regardless from which sources the packages enter the system. We highlighted the six combinations in yellow in Table 8.2. The destinations A2 and E2 are negatively affected the most, because their directly adjacent modules (B2 or D2) are bottlenecks in two ways: Both every package from their adjacent source module (B1 or D1) has to pass through them, and every package to their adjacent destination module (A2 or E2) has to pass through them. To avoid higher transport times, we formulate the following best-practice rule for practitioners: If sufficient space and enough modules are available, no module must have more than one adjacent source/destination module.

Destination	**Sources**								
	B1			C1			D1		
	min*	mean*	SE*	min*	mean*	SE*	min*	mean*	SE*
A2	504	1193.7	4.7	756	1804.4	4.2	1008	2357.5	4.0
A3	756	1589.6	4.3	1008	2091.0	4.1	1260	2460.3	4.0
A4	1008	1925.8	4.0	1260	2351.2	4.3	1512	2584.1	4.3
E2	1008	2364.8	3.9	756	1797.0	4.0	504	1176.1	4.3
E3	1260	2472.1	4.3	1008	2139.3	3.5	756	1644.9	4.8
E4	1512	2617.9	4.3	1260	2359.3	3.9	1008	2017.9	3.7
B5	1008	1923.0	4.1	1260	2347.7	3.9	1512	2580.9	4.0
C5	1260	2250.0	4.2	1008	2061.0	3.9	1260	2221.4	4.2
D5	1512	2609.8	3.7	1260	2356.5	3.9	1008	2037.5	4.3

* in ms

Table 1: Transport times from every source to every destination

9 CONCLUSION AND OUTLOOK

In this publication, we have presented an algorithm that can be used for both small-scaled and large-scaled modular conveyors. We have proven the absence of conflicts, determined the computational complexity, and performed a behavior analysis. The performance of our algorithm is as at least as good as the performance of conventional sorting systems. We are currently in the process of measuring the performance of our algorithm for different layouts and different parameter sets. The results will be published at a later time.

ACKNOWLEDGMENT

This publication was done within the scope of the research project "Vernetzte, kognitive Produktionssysteme (netkoPs)", which was funded by the German Federal Ministry of Education and Research. The authors would like to thank the staff of the Logistics and Distribution Institute (LoDI) at the University of Louisville for their constructive comments on the manuscript.

REFERENCES

1. T. Krühn, S. Sohrt, and L. Overmeyer, "Mechanical feasibility and decentralized control algorithms of small-scale, multi-directional transport modules," *Logistics Research*, Vol. 9, No. 1, p. 147, 2016. [Online]. Available: https://link.springer.com/article/10.1007/s12159-016-0143-x
2. K. R. Gue, K. Furmans, Z. Seibold, and O. Uludağ, "GridStore: A Puzzle-Based Storage System With Decentralized Control," *IEEE Transactions on Automation Science and Engineering*, Vol. 11, No. 2, pp. 429–438, 2014. [Online]. Available: https://doi.org/10.1109/TASE.2013.2278252
3. K. Furmans, F. Schönung, and K. R. Gue, "Plug-and-work material handling systems," *Progress in Material Handling Research*, pp. 132–142, 2010. [Online]. Available: https://digitalcommons.georgiasouthern.edu/pmhr 2010/1
4. S. H. Mayer, "Development of a completely decentralized control system for modular continuous conveyor systems," Dissertation, Karlsruhe Institute of Technology, Karlsruhe, 01.04.2009. [Online]. Available: https://publikationen.bibliothek.kit.edu/1000011463
5. M. Schwab, "A decentralized control strategy for high density material flow systems with automated guided vehicles," Dissertation, Karlsruhe Institute of Technology, Karlsruhe, 24.04.2015. [Online]. Available: http://dx.doi.org/10.5445/KSP/1000047227
6. Z. Seibold, "Logical time for decentralized control of material handling systems," Dissertation, Institut für Fördertechnik und Logistiksysteme, Karlsruhe, 2016. [Online]. Available: http://dx.doi.org/10.5445/KSP/1000057838
7. K.-U. Ventz, "Beitrag zur innovativen Gestaltung von Intralogistik durch Kopplung kleinskaliger Systeme," Dissertation, Leibniz Universität Hannover, Garbsen, 2016. [Online]. Available: http://www.tewiss-verlag.de/katalog/details/?isbn=978-3-95900-083-3
8. I. F. Vis, "Survey of research in the design and control of automated guided vehicle systems," *European Journal of Operational Research*, Vol. 170, No. 3, pp. 677–709, 2006. [Online]. Available: https://doi.org/10.1016/j.ejor.2004.09.020
9. T. Le-Anh and M. de Koster, "A review of design and control of automated guided vehicle systems,"

European Journal of Operational Research, Vol. 171, No. 1, pp. 1–23, 2006. [Online]. Available: https://ssrn.com/abstract=594969

10. L. Qiu, W.-J. Hsu, S.-Y. Huang, and H. Wang, "Scheduling and routing algorithms for AGVs a survey," *International journal of production research: American Institute of Industrial Engineers; Society of Manufacturing Engineers*, Vol. 40, No. 3, pp. 745–760, 2002.
11. O. Uludağ, "GridPick: A High Density Puzzle Based Order Picking System with Decentralized Control," Dissertation, Auburn University, Auburn, Alabama, USA, 2014. [Online]. Available: https://etd.auburn.edu/handle/10415/3984
12. S. Mayer and K. Furmans, "Deadlock prevention in a completely decentralized controlled materials flow systems," *Logistics Research*, Vol. 2, No. 3-4, pp. 147–158, 2010.
13. T. Krühn, "Dezentrale, verteilte Steuerung flächiger Fördersysteme für den innerbetrieblichen Materialfluss," Dissertation, Leibniz Universität Hannover, Hannover, 16.04.2015. [Online]. Available: http://www.tewiss-verlag.de/katalog/details/ ?isbn=978-3-95900-014-7
14. flexlog GmbH, "Dezentral steuerbar – Modulbaukasten mit FlexTechnology," 2019. [Online]. Available: www.flexlog.de/
15. S. Franklin and A. Graesser, "Is It an agent, or just a program?: A taxonomy for autonomous agents," in *Intelligent Agents III Agent Theories, Architectures, and Languages*, J. P. Müller, M. J. Wooldridge, and N. R. Jennings, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 21–35.
16. Beckhoff Automation GmbH & Co. KG, "Flying Motion: XPlanar," 2019. [Online]. Available: https://www.beckhoff.de/xplanar/
17. WEKA BUSINESS MEDIEN GmbH, "Festo – Palettiersystem Motion Cube," 2017. [Online]. Available: https://www.handling.de/video---festo---palettiersystem-motion-cube.htm
18. L. Overmeyer and H. Stichweh, *Vernetzte, kognitive Produktionssysteme: Abschlussbericht*, ser. Berichte des TEWISS Verlags. Garbsen: TEWISS – Technik und Wissen GmbH, 12.12.2017.
19. H. Stichweh, M. Theßeling, S. Sohrt, A. Heinke, and L. Overmeyer, "Intelligent routen, fördern und verteilen: Die Conveyor Matrix für die kognitive Produktion der Zukunft." *25. Deutscher Materialfluss-Kongress mit VDI-Konferenz Routenzugsysteme, TU München, Garching, 17. und 18. März 2016*, pp. 127–142, 2016.
20. S. Sohrt, N. Shchekutin, and L. Overmeyer, "Steuerung von kleinskaligen Fördermodulen," *Logistics Journal Proceedings*, Vol. 2016, No. 10, 2016. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:0009-14-44516
21. C. Uriarte, A.-K. Rohde, and S. Kunaschk, "Celluveyor – Ein hochflexibles und modulares Förder- und Positioniersystem auf Basis omnidirektionaler Antriebstechnik," *18. Magdeburger Logistiktage – Sichere und nachhaltige Logistik*, 2013. [Online]. Available: https://www.iff.fraunhofer.de/content/dam/iff/de/dokumente/publikationen/iff-wissenschaftstage-2013-logistik-tagungsband-fraunhofer-iff. pdf
22. C. Uriarte, H. Thamer, and M. Freitag, "Fördertechnik aus der Zelle," *Hebezeuge und Fördermittel 10*, 2015. [Online]. Available: https://www.technische-logistik.net/sites/default/files/ Fachartikel/HF1015 Thamer 0.pdf
23. H. Thamer, C. Uriarte, and A. Y. Benggolo, "Intelligente Förderzelle: Start-up aus Bremen entwickelt flexibel nutzbare und raumsparende Module," *Hebezeuge und Fördermittel*, Vol. 1-2, No. ISSN: 0017-9442, pp. 56–59, 2018.
24. Itoh Denki Ltd., "Nouveau: MCS Unit, Sorter Module Multidirectional and Multi-Format," 2017. [Online]. Available: http://www.itoh-denki.com/en/les-modules-2/mcs
25. K. C. T. Vivaldini, L. F. Rocha, M. Becker, and A. P. Moreira, "Comprehensive Review of the Dispatching, Scheduling and Routing of AGVs," *A.P. Moreira et al. (eds.), CONTROLO'2014 – Proc. of the 11th Port. Conf. on Autom. Control,* 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-10380-8 48
26. C. W. Kim and J. M. A. Tanchoco, "Conflict-free shortest-time bidirectional AGV routeing," *The International Journal of Production Research*, Vol. 29, No. 12, pp. 2377–2391, 1991.
27. S. Maza, P. Castagna, and IEEE, "Conflict-free AGV routing in bidirectional network," in *ETFA 2001: 8TH IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, VOL 2, PROCEEDINGS*, 2001, pp. 761–764.
28. R. H. Möhring, E. Köhler, E. Gawrilow, and B. Stenzel, "Conflict-free Real-time AGV Routing," *Operations Research Proceedings 2004*, Vol. 2004, 2005. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-27679-3 3
29. T. Stoll, "Dezentral gesteuerter Aufbau von Stetigförderern mittels autonomer Materialflusselemente," Dissertation, Institut für Fördertechnik und Logistiksysteme, Karlsruhe, 02.05.2012. [Online]. Available: http://dx.doi.org/10.5445/KSP/1000028697
30. Institut für Fördertechnik und Logistiksysteme, "KARIS PRO – Autonomer Materialtransport für flexible Intralogistik," Karlsruhe. [Online]. Available: http://karispro.de/Abschlussbericht%20KARIS%20PRO.pdf
31. A. S. Tanenbaum and M. van Steen, *Distributed systems: Principles and paradigms*, 2nd ed. Leiden: Maarten van Steen, 2016.

32. IEEE, “IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems,” 2008. [Online]. Available: https://standards.ieee.org/standard/1588-2008.html
33. J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*. Berlin/Heidelberg: Springer-Verlag, 2006.
34. A. S. Tanenbaum and D. Wetherall, *Computer networks*, 5th ed. Boston, Mass.: Pearson, 2011.
35. J. F. Allen, “Maintaining knowledge about temporal intervals,” Communications of the ACM, Vol. 26, No. 11, pp. 832–843, 1983. [Online]. Available: https://doi.org/10.1145%2F200836.200848
36. S. Luke, C. Cioffi, L. Panait, K. Sullivan, and G. Balan, “MASON: A Multiagent Simulation Environment,” *Simulation*, Vol. 81, No. 7, pp. 517–527, 2005. [Online]. Available: https://journals.sagepub.com/doi/10.1177/0037549705058073
37. Gebhardt Intralogistics Group, “GEBHARDT GridSorter – Modularer Plug & Play Sorter für die Intralogistik – Fit für Industrie 4.0,” 16.05.2014. [Online]. Available: https://youtu.be/--oA-V6emRI
38. D. Jodin and M. ten Hompel, *Sortier- und Verteilsysteme: Grundlagen, Aufbau, Berechnung und Realisierung*, 2nd ed., ser. VDI-Buch. Berlin: Springer Vieweg, 2012.