Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnline CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 51-11 (2018) 1379–1384

Chatting Roles: A Pragmatic Service Resolution Infrastructure for Service Choreography based on Publish/Subscribe

Helmut Zörrer * **Georg Weichhart** *,** **Matthias Plasch** *
Markus Vorderwinkler * **Simon Kranzer** *** **Dorian Prill** ***

* *PROFACTOR GmbH, Steyr, Austria*
(e-mail: helmut.zoerrer@PROFACTOR.at)
** *Pro2Future GmbH, Steyr, Austria*
(e-mail: georg.weichhart@Pro2Future.at)
*** *Salzburg University of Applied Sciences, Salzburg, Austria*
(e-mail: simon.kranzer@fh-salzburg.ac.at)

Abstract: In the following a minimalistic approach for process interoperability, consisting of a topic-based Publish/Subscribe system is presented. It combines an infrastructure and a method guiding the implementation of a basic Service Resolution Infrastructure (SRI) for Service Choreography and Service Interoperability. This SRI can be used to realize resilient collaboration tasks in Industry 4.0 (I4) and Industrial Internet of Things (IIoT) settings without central coordination.

Keywords: Service Choreography, Service Resolution Infrastructure, Industrial Internet of Things, Autonomous Systems, Pragmatic Interoperability, Organizational Interoperability.

1. INTRODUCTION

To be able to reach the vision of a fully flexible production system, reconfiguring itself instantly, interoperability is a key requirement (Weichhart et al., 2016; Tu et al., 2014). Interoperability is required on a technical level allowing to connect heterogeneous production systems and machines. Semantic interoperability is required for exchanging information along the work-flow across multiple systems. Organizational and process interoperability is required, so that individual production steps are synchronized. Currently much work is focusing on technical and semantic aspects of interoperability. Little attention is given to organizational and process-oriented interoperability. In this work we are focusing on the pragmatic aspects and coordination of services along a production process (cf. Bénaben et al., 2015).

In general, two types of service compositions may be distinguished for process interoperability (Weske, 2007):

(1) *Service Orchestration* connects systems a_x and b_y using a central control system (=*orchestrator*). The orchestrator coordinates and integrates processes.
(2) *Service Choreography* supports systems that agree to a choreography before start. Defined roles in this choreography are enacted by implementing systems. This results in the expected overall behavior - without centralized control.
"Dancers dance following a global scenario without a single point of control" (Carbone et al., 2008, p127)

For the realization of service choreography in Industrial Internet of Things (IIoT) settings, a SRI (Service Resolution Infrastructure) is needed (Meissner et al., 2016). In current IIoT system architectures service orchestration based on Business Process Model and Notation (BPMN) workflow management systems or an enterprise service bus (ESB) are predominant (Kopp et al., 2008; Weiß et al., 2016; Hamida et al., 2013). Partially, this is because IIoT Infrastructures have to provide heterogeneous functions e.g. security, software updates for all participants... The disadvantage of using a complex architecture for mediating services and devices is that it represents a bottleneck and single point of failure. In highly distributed environments the need to use a large central master system is inappropriate - hampering also the realization of multiple small autonomous organizational units.

In the following we present a novel methodology to realize a SRI that is only based on a Topic-based Publish Subscribe Infrastructure (TbPSI), which is available for smallest IoT devices. Our simple Chatting Roles Service Resolution Infrastructure (*CrSRI*), enables things to find each other and collaborate without central coordination.

The paper is structured as follows. First we discuss the approach taken, followed by a description of the implementation. We compare our approach with the state of the art before presenting our conclusions and possible future extensions.

2. APPROACH

In certain production systems a high variation of the completion time is present (e.g. when human work is involved). Here global optimizations don't provide an advantage compared to adaptive scheduling with respect to efficiency. However, interoperability between production services and processes can be established by using a

10.1016/j.ifacol.2018.08.335

```
message A_B {
  enum enSubType {
    A_x,  // A->B: Data: data_x, data_x_and_y
    B_y,  // B->A: Data: data_y
    A_z,  // A->B: Data: data_z
  }
  enSubType SubType;
  data_x;
  data_x_and_y;
  data_y;
  data_z;
}
```

Fig. 1. Pragmatic Message Types

message based approach. To reach process and service interoperability, semantic and technical interoperability have to be established first.

Topic-Based Publish/Subscribe Infrastructures (TbPSI) (Eugster et al., 2003) allow systems to publish messages with respect to a certain topic. All systems subscribed to that particular topic will receive those messages.

The semantics of the messages and topics needs to be established first - topic labels, message types and protocols must be known to all systems involved. If a lot of communication between systems is required, many message types and topics must be specified. One solution to limit the number of message types, is the 'Data Type Channel - Message Selector' Hohpe and Woolf (2003) pattern. We describe a pragmatic solution to reduce the number of message types in the next section.

2.1 Pragmatic Message Types - a pattern for less message types

Production systems may take on different roles. Each (physical) system serves different roles when interacting with other systems. The proposed methodology, provides only one message type A_B for all interactions required between roles A and B. To determine the step / state in the interaction protocol, a `SubType` property is available in the message type. All properties required for any of the subtypes have to be available in this message type. Figure 1 shows `SubTypes` $x \ldots z$. In the subtype list messages are sorted as they occur in the choreography. The left side of the message name A_B contains the role (A), that starts the interaction between A and B and we don't define a B_A. (see Figure 1).

The semantics of A_B implies that roles A and B are communicating. A_B holds the set of concepts both A and B need to understand. The advantage is, that much less message types need to be defined. The disadvantage is, that properties are present even if they are not used in the current state. So detailed documentation is required.

2.2 Chatting Roles - methodology

A instance r_z of a role represents a system. In this discussion, two roles are involved - role A and role B. Instances of these roles are a_x and b_y, which will use message type A_B for communication. A major task that every SRI must support is service discovery. In context of roles service discovery can be solved by mediating an appropriate role-player instance of B to an a_x = role discovery.

To implement a role discovery choreography, we mimicked people searching for a service or collaboration using a 'chat-program'. One system starts chatting in a public chat group which covers a relevant topic. Other participants of the group will send a reply message to do some work together. Finally multiple participants could chat on a private level determining the particularities of the collaboration.

Following this, the approach demands an infrastructure with the following communication mechanisms:

- limit communication to a subset of role-instances (= 'public group level' = Community)
- multicast communication from one role-instance to all opposite role-instances (= 'send to opposites')
- individual communication between opposite role-instances that know each other (= 'private message')

Using a TbPSI and some methology, we can implement these communication mechanisms:

- For any role-instance define a proper *Community* to use. Only role instances that use the same *Community* setting will be able to talk with others.
- Any role-instance r_z must have an unique Id. This is defined via configuration field $SystemId_{r_z}$.
- Both roles role A and role B use the proposed *Pragmatic Message Types* - pattern (see section 2.1) for the exchange of messages. So the only concrete message type used in communication of these roles is A_B.
- Any message of type A_B contains a MetaData field to hold a senders $SystemId$ and any r_z sets it to $SystemId_{r_z}$ whenever it sends a message.
- For any message type A_B reserve a least a base topic - path. For simplicity reasons we call this base-path A_B here. But we suggest to precede this path with a name space, to ensure worldwide uniqueness!
- In code specify
  - what message A_B to use
  - what concrete role A or B to play.

  Now we are either an a_x or a b_y.
- Any a_x then must subscribe the following topics:
  - $A\_B/Community/A$
  - $A\_B/Community/A/Systems/SystemId_{a_x}$
- Any b_y must subscribe the following topics:
  - $A\_B/Community/B$
  - $A\_B/Community/B/Systems/SystemId_{b_y}$
- This enables to implement the following functions:
  - a_x.SendPrivate($SystemId_{b_y}$,oMsg) = publish the message to topic $A\_B/Community/B/Systems/SystemId_{b_y}$ (b_y)
  - b_y.SendToOpposites(oMsg) = publish the message to the topic $A\_B/Community/A$ (all a_x)

Figure 2 depicts the proposed usage of topics. Private messages are shown with dotted links. Using the *Pragmatic Message Types* - pattern (see section 2.1) the same message type will be used for all topics. As a consequence of the recipe above, any a_x now is listening to it's private message topic and to a global message topic that all As are listening too and vice versa the same is true for any b_y. Multicast messages are realized using 2 separated topics, one for each role. Usually no A sends to other As. This prevents unnecessary or cycle-messages per default.

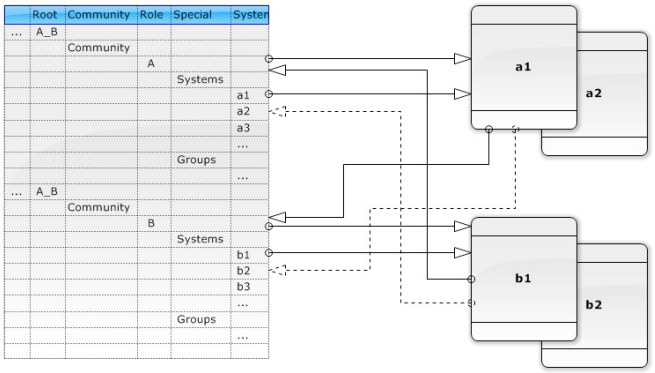

Fig. 2. Chatting Roles - Proposed Usage of Topics for Roles

Role-instances can only talk to each other, if they all have configured the same *Community*. Sending private messages to other opposite roles demands knowledge of the opposite role's private *SystemId* first. This data can be transmitted using appropriate role interactions.

2.3 Testing Chatting Roles fitness to fulfill some Service Interaction Patterns

Service Interaction Patterns are used to benchmark languages for the ability to express advanced conversations(Weske, 2007). We assume that an algorithm/system supporting these patterns, enables the implementation of a service choreography system. We examined whether using topic-based publish/subscribe and our proposed method above, can be used as a base to implement the required patterns (Weske, 2007):

Send *represents a one-way interaction between two participants seen from the perspective of the sender*:
If an a_x want's to send a Message to a b_y, with $SystemId_{b_y}$, it publishes to $A\_B/Community/B/Systems/SystemId_{b_y}$ and vice versa b_y.

Receive *describes a one-way interaction between two participants seen from the perspective of the receiver*:
Any a_x has subscribed it's private topic $A\_B/Community/A/Systems/SystemId_{a_x}$. So it can receive individual messages and vice versa any b_y.

Send / Receive *a participant sends a request to another participant who then returns a response message*:
Can be realized by placing correlation information inside the message (Weske, 2007)

One-To-Many-Send *a participant sends out several messages to other participants in parallel*:
Any a_x can send to $A\_B/Community/Systems/B$ which any b_y has subscribed.

One-From-Many-Receive *one participant waits for messages to arrive from other participants, and each of these participants can send exactly one message*:
Can be solved on the receiver side, by only accepting e.g. the first or last message of any sender in the context of this One-From-Many-Receive within a special time frame.

other *One-To-Many Send/Receive, Racing Incoming Messages, Multi-Request. . .*:
These patterns can also be solved using special message fields, timeouts or special behavior on a_x resp. b_y.

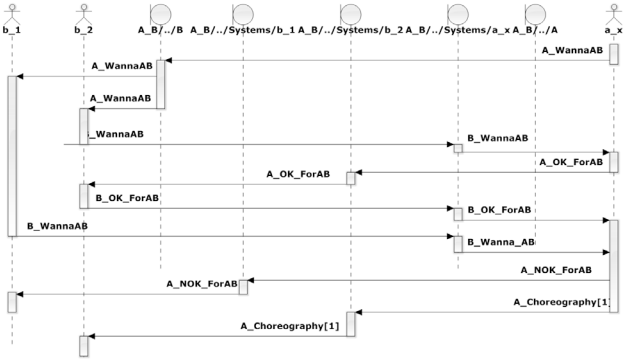

Fig. 3. WannaAB - how a_x and b_y find each other

In the following a service discovery protocol using the *ChattingRoles* Infrastructure is described. *An a_x want's to be mediated to one dedicated b_y to realize the choreography A_B together then (˜Discovery)*

2.4 WannaAB - a basic Discovery Choreography using Chatting Roles

Figure 3 shows the suggested flow of messages and topic usage. a_x starts using a_x.SendToOpposites(A_WannaAB). As prepared by our architecture, this will be delivered to all Bs. a_x may repeat the search in a configurable interval. Any concrete b_y willing to perform A_B may wait now for multiple As to choose the best partner. It may rank As on nearest location, or cheapest price, etc. However, it finally answers b_y.SendPrivate($SystemId_{a_x}$, B_WannaAB).
a_x can also analyze multiple answers for best fitting candidates. Finally it decides, if it really want's to realize A_B with this concrete b_y. It now reacts with a_x.SendPrivate($SystemId_{b_y}$, A_OK_ForAB)
(or A_NOK_ForAB). b_y receiving A_OK_ForAB can decide to agree or regret the communication and answer either B_OK_ForAB or B_NOK_ForAB to a_x. Finally a_x receiving B_OK_ForAB starts the private conversation - the *Payload-Choreography* with the concrete given private $SystemId_{b_y}$.
a_x and b_y can execute their A_B choreography as long as needed. If choreographic faults or timeouts are detected they may fallback to discovery again.
As an alternative to sending messages after mediation a_x can provide (REST) services to b_y and vice versa, if some message-data contains hints on how to use these services (e.g. URIs, IP-address, . . .). What solution to choose for payload communication only depends on architectural preferences.

3. IMPLEMENTATION

The chatting role system has been implemented in a concrete scenario, where automated guided vehicle (AGV) implement a transport service between an assembly and a manual packaging area (see Figure 4). An initial assembly step takes place, resulting in a part being available after $T_s = 4$ time ticks. An AGV picks up the part and transports it to an available packaging station, where $T_p = 3$ time ticks are needed to complete this simple process. There are no

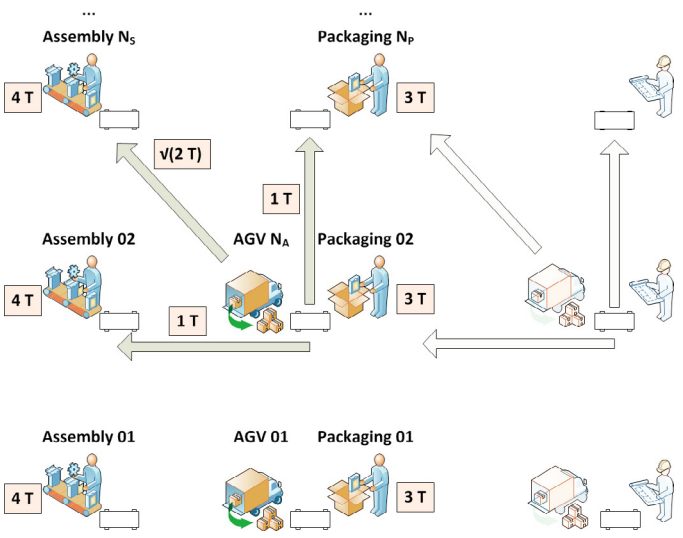

Fig. 4. AGV Transport Scenario

```
message AGVPickup_ProducerTransportNeed {
  MetaData MetaData;
  enum enSubType {
    Protocol;                 // Delegating 'WannaAB things'
    AGV_DeliverPart;          // AGV->Producer
    Producer_PartDelivered;   // Producer->AGV [PartData]
  }
  enSubType SubType;
  PartData Part;
}
message AGVDelivery_ConsumerPartNeed {
  MetaData MetaData;
  enum enSubType {
    Protocol;             // Delegating 'WannaAB things'
    AGV_TakePart;         // AGV->Consumer [PartData]
    Consumer_PartTaken;   // Consumer->AGV
  }
  enSubType SubType;
  PartData Part;
}
message MetaData {
  string Id;              // message id
  string SystemRole;      // what role plays the sender
  string SystemId;        // who sent the message
  string Topic;           // topic this message was sent to
  string SessionId;       // communication context
  float  Lat;             // location-info
  float  Long;            //  e.g. to prefer nearest device
  float  AvailableAt;     // time(seconds) when available
  int32  Priority;
  enum enProtocolType {// on any X_Y naming remains A,B!
    Default;              //
    A_WannaAB;            // A initiates discovery for Bs
    B_WannaAB;            // all Bs willing to play respond
    A_OK_ForAB;           // finally A accepts one B
    A_NOK_ForAB;          // or A regrets a B
    B_OK_ForAB;           // finally B accepts one A
    B_NOK_ForAB;          // or B regrets to play with A
  }
  enProtocolType PROTOCOLTYPE;
}
```

Fig. 5. GPB Message Types in Scenario

buffers at stations, implying that not working transports block the overall processes. AGVs driving times are shown in figure 4. The minimum transport time $Tmin_d$ is given as 1 time tick. We depict the usage of message types (Figure 5) roles communities and choreography in Figure 6.

We implemented our approach in a .NET environment using Google Protocol Buffers (GPB `https://developers.google.com/protocol-buffers/`) for messages. As TbPSI infrastructure we used RabbitMQ and Advanced Message Queuing Pro-

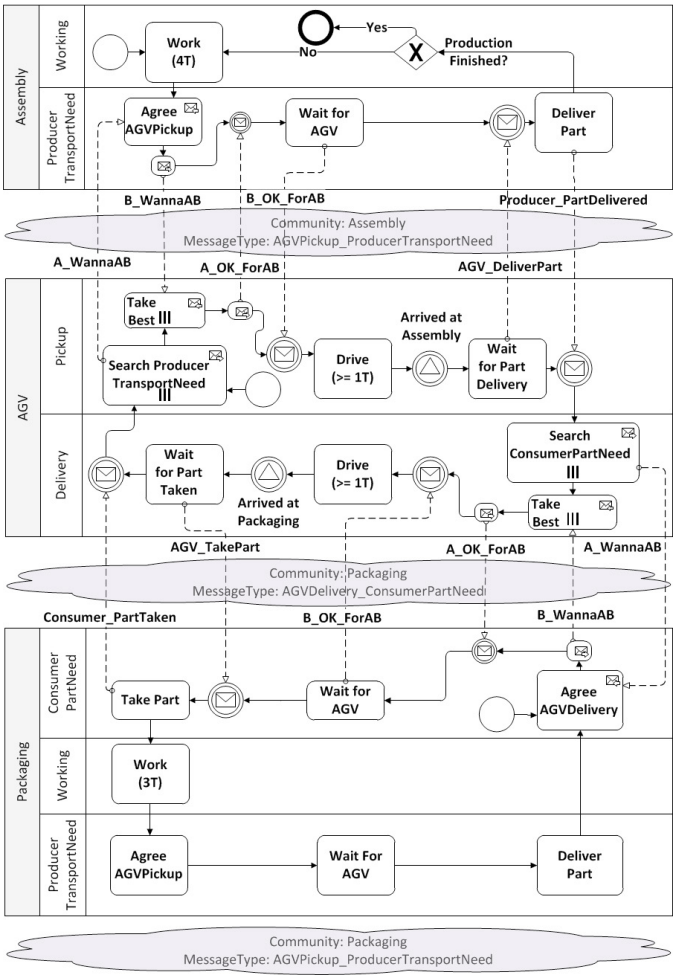

Fig. 6. BPMN - Representation of Choreography

tocol (AMQP). We developed a test-framework to start and stop instances of assemblies, packages and AGVs as needed (mapping 1 second to 1 time tick). For our environment we used an initial time factor t_f of 100ms. We then calculated the following variables as:

- $WAIT\_FOR\_MULTIPLE\_OPPOSITES = t_f * 2$ = Maximum time to wait for opposites
- $REPEAT\_MULTIPLE\_WAIT = t_f * 3$ = interval to repeat searching
- $WAIT\_FOR\_RESPONSE = t_f * 4$ = Maximum time to wait for a response for $WannaAB$

This first prototype was used to evaluate the functionality of the service discovery mechanism. It worked as expected in the $CrSRI$ framework. In a second step the $WaitForMultipleMessages$ function was implemented, to be used by the searching role to filter and rank messages received from agreeing partners. This enabled us to implement dispatching rules on AGV searcher side (e.g. prefer shortest path). So instead of using some external control to mediate opposites, in our infrastructure this can be done by the devices themself.

In a next step we implemented a preventive transport (Pt) approach to reduce transport times. With Pt intelligent assemblies and packagings can contact AGVs while they are still working. As an additional information they also publish their finish time (MetaData 'AvailableAt'). AGVs

Table 1. Computational Results

	Rt		Duration [T]		#Msg		#Msg/Second		Max Msg/Second	
Experiment	avg.	stdev.	avg.	stdev.	avg.	stdev.	avg.	stdev.	avg.	stdev.
2A	3,48	0,30	70,62	3,60	1134,20	47,81	5,31	0,28	26,70	2,71
2ASp	3,01	0,27	71,18	2,25	1293,50	35,21	6,05	0,26	27,70	2,75
2ASpPt	2,07	0,17	60,47	1,12	1130,90	35,78	6,22	0,21	31,80	4,49
2ASpPt(50ms)	2,92	0,33	68,50	2,19	1772,50	119,94	8,61	0,45	35,90	3,41
2ASpPt(200ms)	3,34	0,09	69,39	0,72	1107,50	28,10	5,31	0,10	27,90	3,03
3A	3,05	0,19	67,15	2,55	1854,40	54,20	9,20	0,33	31,00	3,19
3ASp	2,65	0,24	66,90	2,09	2000,90	64,14	9,97	0,40	32,70	3,91
3ASpPt	1,31	0,25	51,39	1,31	1551,10	108,37	10,06	0,80	35,40	3,02

Fig. 7. Gantt Output for a Dynamic Scenario in Development Environment

that support Pt only accept opposites, if driving time is less than the given part finish time.
Using $N_s = 3$ assembly stations, each producing $P_s = 10$ parts and $N_p = 3$ packaging stations, we conducted different experiments and executed each of them in 10 runs. To understand whether things have improved, we implemented the following simple rating function.
$Rt = \sum_{n=1}^{N_s * P_s}(tCompletion_n - (T_s + T_p + Tmin_d))/(N_s * P_s)$
Rt calculates the average transportation waste per part for a solution. The range of Rt for each experiment is shown in Table 1 (Experiment Naming convention: e.g. $2ASpPt(50ms)$: $2A = 2$ AGVs are used. $Sp =$ AGVs prefer shortest paths. $Pt =$ assembly and packaging support preventive transport. $(50ms) =$ instead of $t_f =$ 100ms we used 50ms).

Our experiments revealed, that preventive transport is in our scenario more efficient than using additional AGVs. Timings have to be chosen carefully, since timings have a high impact on the overall performance. The longer roles wait for receipt the longer the total dispatching process will take. But if roles do not wait long enough, they may miss a better fitting opposite, and more messages have to be processed by the broker too. Max Msg/Second of about 40 and average message size measured in communication of 120 bytes leads to a network traffic of about 5kB/s for our scenario. In Figure 7 we show results from a more dynamic scenario where entities involved use different strategies resp. software versions. In this scenario we configured only the first packaging and the first assembly to support preventive transport. This leads to less Blocking times on Assembly_1 and less Waiting time on Packaging_1). Additionally we started a third AGV_3 after a few seconds. And we closed down Packaging_3 later during production, without production suffering.
Additional areas of production after packaging (e.g. shipping) can be supported, by using the same message types but additional communities (As indicated at the end of Figure 6). If an AGV shall serve in another production area (or in another company), the only things to change are the two communities used for pickup and delivery (and the broker url). If a part shall not visit all production areas, the AGV can take the concrete work plan from the part it transports. Either the AGV always carries the same part for production, or it may return to it's base community after delivery, or it may always search in multiple/all communities for transport jobs.

4. RELATED WORK

In this section we compare our approach to related works. First two Enterprise Service Bus (ESB) based engines are presented. This is followed by realtime and IoT focused approaches.

The main focus of the *ChorSystem* (Weiß et al., 2016) framework is to manage the complete life cycle of choreographies, from modeling workflows through deployment to execution and monitoring. A central ESB based workflow engine is used to mediate services of participants registerd at the global *ChorSystem* service registry . The ESB based CHOReOS/CHOReVolution (Hamida et al., 2013) project enables the execution of large-scale IoT service choreographies and discovery of services during runtime. Both approaches focus more in networking of larger infrastructures, than in supporting multiple autonomous cells. Trends of resource discovery in IoT are analyzed in (Vandana, 2016). Discovery is incomplete without retrieving information and ranking it up. "IoT devices are mostly power constrained, so to save power, they wake up or become active only when required to perform a specific function"(Vandana, 2016, p3085) and analyze the challenges when using IoT resource discovery in constrained application protocol (CoAP). Efficient Ranking in CoAP seems not to be supported. Other limitations in CoAP-RD based resource discovery are lacking scalability and unclear specification (Vandana, 2016). Using *CrSRI* we can rank things for mediation. To save energy chatting only sometimes may be a solution. And dedicated brokers may be a solution to solve the high communication demand. The huge amount of IoT devices will still remain a hard challenge. *CrSRI* may be more useful for medium-sized cooperative CPPS scenarios. Kothmayr et al. (2016) propose a real-time service oriented architecture (rtSOA) for instant service choreographies, but service discovery was out of the scope in the TDMA based approach. Publish/subscribe architectures can also be used in real-time settings (Paikan et al., 2015). Self assembling robotic systems (Liu and Winfield, 2012) describes robots recruiting strategies and the behaviors they use. They use infrared communication with a limited range of maximal 150cm. We assume that their recruitment state could be a good candidate to be implemented with *CrSRI*, if finding bots in a larger area is of interest. The AGV scenario has some analogy to foraging robots in collective robotic area (Winfield, 2009). In foraging a robot usually is in one of 4 states: *searching*, *grabbing*, *homing* or *depositing*. For

AGVs *searching* can be seen as the task to 'search an assembly that has a part to carry' and *homing* is the task to 'search for a free packaging'. Whereas foraging robots usually only sense their near environment for food, in our scenario food is intelligent enough to tell where it can be found. So in our AGV scenario search tasks can be done using $CrSRI$ and $WannaAB$. The search discovery scope is defined by the community and roles the AGVs use.

5. CONCLUSIONS AND FUTURE WORK

We implemented a pragmatic Chatting Roles Service Resolution Infrastructure ($CrSRI$) and evaluated its performance. Our evaluation showed that our simple SRI can be used to enable resilient cooperation of IIoT devices. Process interoperability between the different systems is supported, but requires technical, and semantic interoperability by implementing a fixed set of the message types first. The presented approach is not only suited for IoT but also for socio-technical systems (with human work and completion time variations). The system allows human operators to finish their tasks without imposing a global schedule. We are working on the following future research:

- For publish/subscribe without topics (e.g. *Google's Nearby API*) or broadcast networks, topics can be surrogated if sending roles set the message MetaData Topic field. Receiving roles must ignore unwanted types of message and filter the remaining by means of this Topic field = process only 'subscribed' messages.
- Implement software update for devices. Use e.g. $SoftwareUpdater\_RoleX$ messages to deliver the new software and let devices sense on that messages.

ACKNOWLEDGEMENTS

The research described in this paper has been partially funded by the European Union and the state of Upper Austria within the strategic economic and research program Innovative Upper Austria 2020 and the projects "Smart Factory Lab" and "DigiManu".

First basic considerations on the use of topics in TbPSI were made in the Intelligent Maintenance Planner project (Kranzer et al., 2017). Additional thanks go to Alexander Hämmerle for fruitful discussions.

REFERENCES

Bénaben, F., Mu, W., Boissel-Dallier, N., Barthe-Delanoe, A.M., Zribi, S., and Pingaud, H. (2015). Supporting interoperability of collaborative networks through engineering of a service-based mediation information system (mise 2.0). *Enterprise Information Systems*, 9(5-6), 556–582. doi:10.1080/17517575.2014.928949.

Carbone, M., Honda, K., and Yoshida, N. (2008). Theoretical aspects of communication-centred programming. *Electronic Notes in Theoretical Computer Science*, 209, 125 – 133.

Eugster, P.T., Felber, P.A., Guerraoui, R., and Kermarrec, A.M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 114–131. doi:10.1145/857076.857078.

Hamida, A.B., Kon, F., Lago, N., Zarras, A., Athanasopoulos, D., Pilios, D., Vassiliadis, P., Georgantas, N., Issarny, V., Mathioudakis, G., Bouloukakis, G., Jarma, Y., Hachem, S., and Pathak, A. (2013). Integrated choreos middleware - enabling large-scale, qos-aware adaptive choreographies. URL `https://hal.inria.fr/hal-00912882`.

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kopp, O., van Lessen, T., and Nitzsche, J. (2008). The need for a choreography-aware service bus. In *The 3rd European Young Researchers Workshop on Service Oriented Computing (YRSOC 2008)*, 30–36.

Kothmayr, T., Kemper, A., Scholz, A., and Heuer, J. (2016). Instant service choreographies for reconfigurable manufacturing systems - a demonstrator. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1–8. doi:10.1109/ETFA.2016.7733606.

Kranzer, S., Prill, D., Aghajanpour, D., Merz, R., Strasser, R., Mayr, R., Zoerrer, H., Plasch, M., and Steringer, R. (2017). An intelligent maintenance planning framework prototype for production systems. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, 1124–1129. doi:10.1109/ICIT.2017.7915520.

Liu, W. and Winfield, A.F.T. (2012). *Distributed Autonomous Morphogenesis in a Self-Assembling Robotic System*, 89–113. Springer Berlin Heidelberg.

Meissner, S., Debortol, S., Sperner, K., Magerkurth, C., and Dobre (2016). Internet of things architecture - a project deliverable d2.3 - orchestration of distributed iot service interactions. `http://www.meet-iot.eu/deliverables-IOTA/D2_3.pdf`. Last visited 2016-09-01.

Paikan, A., Domenichelli, D., and Natale, L. (2015). Communication channel prioritization in a publish-subscribe architecture. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, 41–45. doi:10.1109/SEARIS.2015.7854100.

Tu, Z., Zacharewicz, G., and Chen, D. (2014). A federated approach to develop enterprise interoperability. *Journal of Intelligent Manufacturing*, 27(1), 11–31. doi:10.1007/s10845-013-0868-1.

Vandana, C. (2016). Study of resource discovery trends in internet of things (iot). *Int. J. Advanced Networking and Applications*, 08(03), 3084–3089.

Weichhart, G., Molina, A., Chen, D., Whitman, L., and Vernadat, F. (2016). Challenges and current developments for sensing, smart and sustainable enterprise systems. *Computers in Industry*, 79, 34–46. doi:10.1016/j.compind.2015.07.002.

Weiß, A., Andrikopoulos, V., Hahn, M., and Karastoyanova, D. (2016). ChorSystem: A Message-based System for the Life Cycle Management of Choreographies. In *On the Move to Meaningful Internet Systems: OTM 2016*, 503–521. Springer-Verlag.

Weske, M. (2007). *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York, Inc.

Winfield, A.F. (2009). *Foraging Robots*, 3682–3700. Springer New York, New York, NY.