Distributed tracing of OPC UA method calls

C. Mayr-Dorn, B. Pereszteghy, J. Holzweber, M. Mayrhofer

With increasing digitalization, shopfloor architectures transition to service-oriented, distributed layouts in which the complexity of monitoring communication between systems becomes a major challenge. Distributed tracing assists in establishing causality and hence supports the analysis of latency aspects, wrongly configured communication endpoints, and bottlenecks. In this paper, we present a first feasibility study, which investigates to what extent it is possible to trace OPC UA method calls in a distributed manner using the Zipkin framework. We show how this standard can be used in conjunction with the Eclipse Milo OPC UA open source stack and how it can be integrated into our industry demonstrator "Factory in a Box".

Keywords: distributed production systems; distributed tracing; OPC UA; method invocation; correlation propagation

Verteiltes Nachverfolgen von OPC UA Methodenkommunikation.

Mit zunehmender Digitalisierung werden Shopfloorarchitekturen zunehmend verteilt und serviceorientiert. Somit wird die Komplexität der Überwachung der Kommunikation zwischen den teilnehmenden Systemen zu einer immer größeren Herausforderung.

Hierbei hilft die verteilte Ablaufverfolgung bei der Ermittlung der Kausalität und unterstützt somit die Analyse von Latenzaspekten, falsch konfigurierten Kommunikationsendpunkten und Kommunikationsengpässen.

In diesem Artikel präsentieren wir eine erste Machbarkeitsstudie, die untersucht, inwieweit es möglich ist, OPC UA-Methodenaufrufe mithilfe des Zipkin-Frameworks verteilt zu verfolgen. Wir zeigen, wie dieser Standard in Verbindung mit dem Eclipse Milo OPC UA Open Source Stack verwendet und in unserem Industriedemonstrator "Factory in a Box" integriert werden kann.

Schlüsselwörter: verteilte Produktionssysteme; verteiltes Nachverfolgen; OPC UA; Methodenaufruf; Korrelationsweiterleitung

Received May 10, 2021, accepted June 29, 2021, published online July 15, 2021

1. Introduction

As digitalization advances, shopfloor architectures transition to service-oriented, distributed layouts. As a communication protocol, OPC UA is introduced at multiple hierarchy levels, e.g., cell, station, substation, and even component (see, for example, the VDMA R+A OPC UA Demonstrator [17]). The two trends together create highly distributed CPPS, in which the complexity of monitoring communication between systems becomes a major challenge.

Distributed tracing assists in understanding which communication and action on the shopfloor is the result of what other action. This, in turn, helps establish causality and hence supports the analysis of latency aspects, wrongly configured communication endpoints, and bottlenecks.

As a centralized schema for monitoring is typically only feasible when a single vendor provides an integrated solution, yet with elements coming from multiple vendors, a more lightweight approach is more feasible.

In this paper, we present a first feasibility study to what extent it is possible to trace OPC UA method calls in a distributed manner. To this end, we propose to use the widely used Zipkin standard for tracing information modeling and trace correlation information propagation. We show how this standard can be used in conjunction with the Eclipse Milo OPC UA open source stack and integrated into our industry demonstrator: "Factory in a Box". We also discuss the various ways OPC UA interaction affordances can be instrumented, and the open issues deriving from a communication environment that aside from method calls also uses data access, monitored items, and events.

1.1 Motivating scenario

Our lab-scale production cell, as part of a university-funded demonstrator project Factory in a Box[1] (FIAB), aims at illustrating the concepts that enable flexible production. Our particular demonstrator can customize the drawings on a piece of paper at multiple plotting stations. The production cell consists of the following machine types: input stations that provide pallets with paper, plotters that load the pallets and draw images in one color each, turntables that transport the pallets between plotters, and finally, output stations where the finished product (i.e., paper) is placed.

Communication between the machines is purely based on OPC UA. Plotters and output stations are designed and programmed with the IEC 61499[2] industry standard using the open source IDE Eclipse 4diac [13] and the respective FORTE runtime environment that builds on the Open62541 OPC UA [5] server. The turntables (and virtual

[1] https://github.com/jku-isse/factory-in-a-box.

[2] http://www.iec61499.de/.

Part of this work has been supported by LIT-2019-8-SEE-118, LIT-ARTI-2019-019, and LIT Secure and Correct Systems Lab funded by the state of Upper Austria, as well as *Pro^2 Future* (FFG under contract No. 854184). *Pro^2 Future* is funded within the Austrian COMET Program—Competence Centers for Excellent Technologies — under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

Mayr-Dorn, Christoph, Johannes Kepler University Linz, Altenbergerstraße 69, 4040 Linz, Austria (E-mail: christoph.mayr-dorn@jku.at); **Pereszteghy, Benno,** Johannes Kepler University Linz, Linz, Austria; **Holzweber, Jan,** Pro2Future GmbH, Linz, Austria; **Mayrhofer, Michael,** Pro2Future GmbH, Linz, Austria

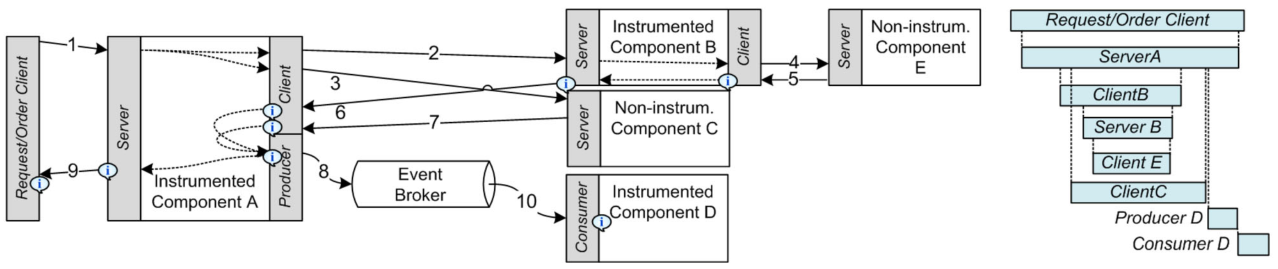


Fig. 1. Example control flow through a distributed system (left), with instrumented and non-instrumented components, using method calls and events (full lines); displaying component internal control flow with dashed lines. Info icons mark the instrumentation points where trace information is sent to the central trace collector (e.g., a Zipkin server). The diagram to the right displays the resulting spans and their relations. Span names identify where the tracing information has been captured

twins of the prior introduced machines) are implemented in Java, using the Eclipse Milo OPC UA framework. Hence, despite the toy character of the setup, the used software and communication infrastructure is industrial grade. We use Lego Mindstorm EV3-based PLCs as a basis as this enables rapid and cheap machine prototyping without the need to test individual subcomponents such as motors, actuators, and sensors. Also, it ensures their seamless integration.

In the scope of this article, we are especially interested in tracking individual orders through the FIAB shopfloor, and hence, need to trace how the machines communicate for fulfilling each order. Given the highly dynamic layout, one critical support is quick detection of incomplete or incorrect wiring of communication paths between machines, production deadlocks, identify the potential for parallelizing activities, and understanding to what extend the production time of each order is determined by the machine one the one hand and coordination needs among the machines on the other hand. In short, we need to understand how the behavior of machines (i.e., method calls, etc.) is causally related to the individual production orders.

2. Distributed tracing background

This section provides only a brief introduction to the concepts used in distributed tracing. The interested reader finds a more thorough discussion in [11].

The idea of distributed tracing is tracking how a batch, an order, a single command, or event is processed by a (distributed) production system. The ripple effect of further requests or events is then captured in a "trace". Figure 1 (left) depicts a simplified control flow graph through an example system. Different approaches are possible to know which request/events belong to the same trace (see also Sect. 5). The state-of-the-art approach to obtain such a correlation is using metadata propagation of a trace identifier. This implies the instrumentation of the various system components with tracepoints. These tracepoints are typically placed where a request or event enters a component, respectively, where a response or event is returned.

A key aspect of distributed tracing is capturing the causality amongst requests, i.e., how is the sending of one request was caused by the reception of another, earlier request. In the example flow it would be potentially difficult to assess based only on timestamps and trace identifier information alone whether request 4 occurred due to request 2 or due to request 3. To this end, trace metadata propagation typically also includes a parent request identifier. For example, both requests 2 and 3 will have request 1 as a parent, request 4 will have request 2 as a parent.

Tracepoints by themselves just represent events in a system. Even with causality relationships among these events (that then build up an acyclical directed graph), users find it usually hard to easily interpret such trace information. Most modern tracing approaches thus utilize "spans" to group events that meaningfully describe a component's processing activities. These spans build up a tree where lower spans describe the more detailed, fine granular processing. Figure 1 (right) depicts the span hierarchy for the depicted control flow on the left. Note that the uppermost client span (i.e., the root span), is complete before the end of *Consumer D* and potentially even *Producer D* span due to their asynchronous execution.

Ultimately, span-based distributed tracing such as Zipkin foresees the sending of spans to the central span collection server (e.g., a Zipkin server) upon the finishing event of a span (indicated in Fig. 1 with the info icons).

2.1 Zipkin

In Zipkin, a span entry defines the following properties which we briefly introduce. We discuss a more detailed mapping to OPC UA in Sect. 3.

– traceId: same random id for all spans that belong to the same trace.
– name: the activity or operation this span represents, e.g., an OPC UA skill or the OPC UA method name if available.
– parentId: empty if this is the root span, otherwise parent span's *id*.
– id: uniquely identifies this span
– kind: additional, optional information to allow interpretation of *timestamp*, *duration*, and *remoteEndpoint*. We discuss the implications of the possible values (CLIENT, SERVER, PRODUCER, and CONSUMER) below.
– timestamp: an as accurate as possible measure representing the start of this span.
– duration: how long this span lasted. The interpretation depends on the span's *kind*.
– localEndpoint: describes the networked component that created this span in terms of a *serviceName*, *ipv4* or *ipv6* address, and *port* number.
– remoteEndpoint: describes the networked component to which a causal relation exists in the scope of this trace. The span's kind determines how to interpret the *remoteEndpoint*.
– annotations: capture additional events that occur between the start and end of this span. Each annotation consists of a timestamp and a string describing the event, e.g., for capturing when an error occurred, perhaps an internal state transition happened, etc.

– tags: provide additional endpoint or vendor-specific information such as version information, query string, request parameters, etc. as a set of key-value pairs.

Aside from *traceId*, and *id*, (and *parentId* for child spans), no other properties are strictly mandatory.

CLIENT describes a span from the view of a component that requests some service from a remote component (described in the span's *remoteEndpoint*), capturing in the span's *timestamp* the instant the request was sent and measuring in the span's *duration* the time until a response is received.
SERVER describes a span from the view of a component that receives a service request from a remote client (described in the span's *remoteEndpoint*), capturing in the span's *timestamp* the instant the request was received, and measuring in the span's *duration* the time it takes to send back a response.
PRODUCER describes the span from the view of a component that dispatches an event to an event broker (described in the span's *remoteEndpoint*) without any knowledge of who the receiving consumers will be. The span's *timestamp* indicates the time of handing the event over to the broker. The span's *duration*, if present, measures the time between handing over the event, and the event being actually sent to consumers.
CONSUMER described the span from the view of a component that receives an event from an event broker (described in the span's *remoteEndpoint*) without any explicit knowledge of who the sender of the event is. The span's *timestamp* indicates the time of obtaining the event from the broker, the *duration*, if present, measuring the time between the reception by the event broker and the start of being processed by the consumer component.

Note that in the case of CLIENT and SERVER the span's *duration* is typically more meaningful as it describes a component's/service's/skill's latency as well as network latency, whereas the span's *duration* for PRODUCER and CONSUMER only measures the latency of the event broker and requires application-specific knowledge to measure any type of service/skill latency.

3. Mapping Zipkin to OPC UA

Two aspects need consideration when applying distributed tracing to OPC UA. First, the propagation of trace meta-data, and second, the use of spans.

3.1 Propagating trace correlation information

Zipkin uses the B3 header specification[3] for which mappings to HTTP headers, Apache Kafka record header, gRPC ASCII headers, or JMS headers already exist.

We propose to encode the B3 equivalent information in an OPC UA method call's *additionalHeader* element defined in the OPC UA standard.[4]

At the moment, no standardization exists yet that would manage how multiple, independent header extensions can coexist in the additional header element. Hence, the exact injection and extraction procedure of the B3 header from OPC UA method calls is very likely to change; the encoding of the B3 information itself, however, will not be affected by this.

```
<B3
    traceId="80f198ee56343ba864fe8b2a57d3..."
    spanId=\e457b5a2e4d86bd1"
    parentSpanId="05e3ac9a4f6e3b90"
/>
```

In order to provide the B3 information for injection into the actual OPC UA additionalHeader element, we needed to extend a few classes in the Eclipse Milo stack (v0.3.8) as the header is not sufficiently exposed via the stack's API. Specifically, on the client-side we extended the OpcUaClient and the UaStackClient to enable handing over and insertion of B3 information into the header and the resulting dispatching of span information to the Zipkin server upon method call completion. On the server-side, we extended the ServerSymmetricHandler, StackServer, SessionManager, and Session to extract the B3 header information from the request, and thereby enable any server logic to pass on the B3 information for further propagation, and likewise, the dispatching of span information to the Zipkin server. Note that the client can send span information to the Zipkin server also when invoking a method on a non-instrumented server, as it doesn't rely on header information in the invocation response. We additionally relied on the Brave library[5] for the actual dispatching of the span information to Zipkin.

3.2 Implications of OPC UA usage

Contemporary web-based systems primarily use a procedure call-centric architecture where a request is typically expecting a response with the desired content. In contrast, with OPC UA, method requests are expected to finish as quickly as possible, thus being most often used merely as triggers to longer running actions on the server-side. The result and success of such actions are then inspected using separate method calls, data access operations, monitored items, or events.

Consequently, even when instrumenting OPC UA method calls and creating spans of kind "SERVER", respectively "CLIENT", this will result in a span structure more similar to spans of kind "PRODUCER" and "CONSUMER". We demonstrate this behavior in Sect. 4 where we also discuss the opportunity to lift tracing instrumentation into the component logic to obtain a more easily understandable span hierarchy.

Aside from method invocations, requests to read and write an OPC UA node's value via OPC UA Data Access can be equally instrumented with B3 header extension information. Here, however, we have to differentiate the direction of the correlation information flow for reads and writes. A write can be interpreted as a form of request. Note, that this will not be the case for every data item; hence such instrumentation only makes sense where a write is expected to trigger some behavior on the server-side that propagates correlation information and is traced. Here, the request header will contain the relevant B3 correlation information. Upon dispatching the write request, the OPC UA client may then create a span of kind "PRODUCER", and the server should create a span of kind "CONSUMER" whenever that node is next accessed locally. A read, in contrast, can be used as a mechanism to providing a return value from a long-running action, or a trigger to execute some next steps on the client side. In both cases, the B3 header in the response will be meaningful to propagate context from server to client. Creating of spans is reversed here. Whenever the node value changes at the server, the server may create it as a span of kind "PRODUCER", and the OPC UA client will create a span of kind "CONSUMER" whenever it reads

[3]https://github.com/openzipkin/b3-propagation.

[4]https://reference.opcfoundation.org/v104/Core/docs/Part4/7.28/.

[5]https://github.com/openzipkin/brave.

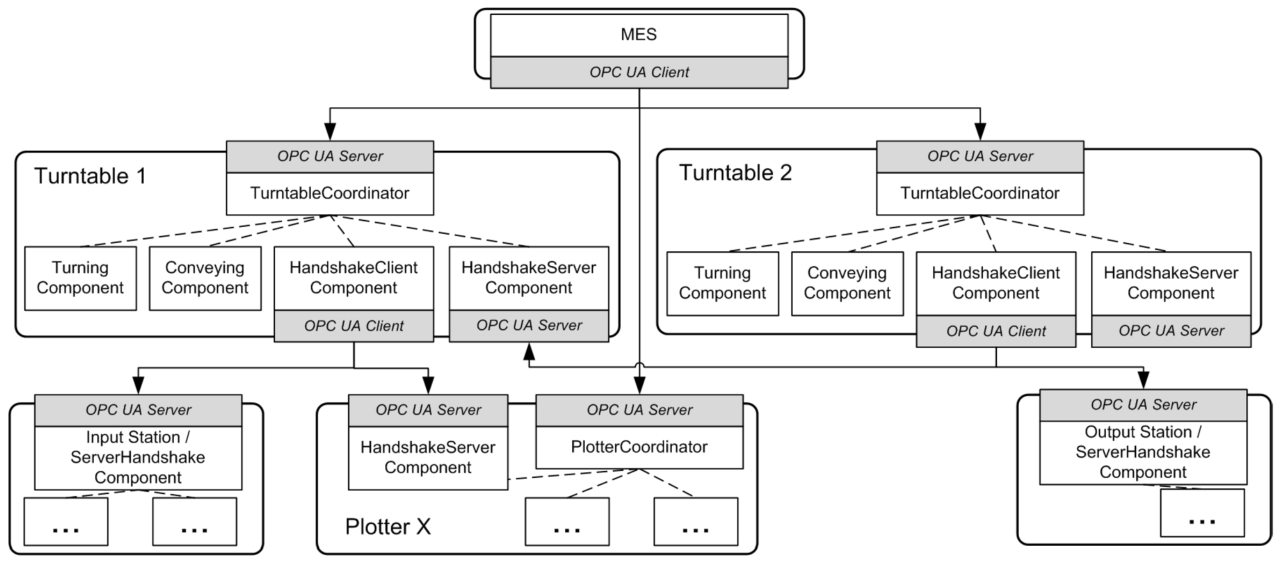


Fig. 2. Conceptual architecture excerpt of the Factory in a Box showing OPC UA method calls (full lines), and machine internal communication paths (dashed lines)

the value. Note that in the former case, the client could also retain the B3 propagation information from the initial long-running action trigger, thus not relying on any B3 response header information. For both read and write, however, there is more instrumentation needed on the server-side than for method invocations, as the server has to maintain B3 information for every node that is used for correlation propagation. The same read-specific aspects hold for propagating correlation information when using monitored items.

Transparent instrumentation of OPC UA events is more difficult, as, at the time of writing, no standardized mechanism for including meta-information in an event is available. Hence, any correlation information needs to be explicitly modeled in the event data model.

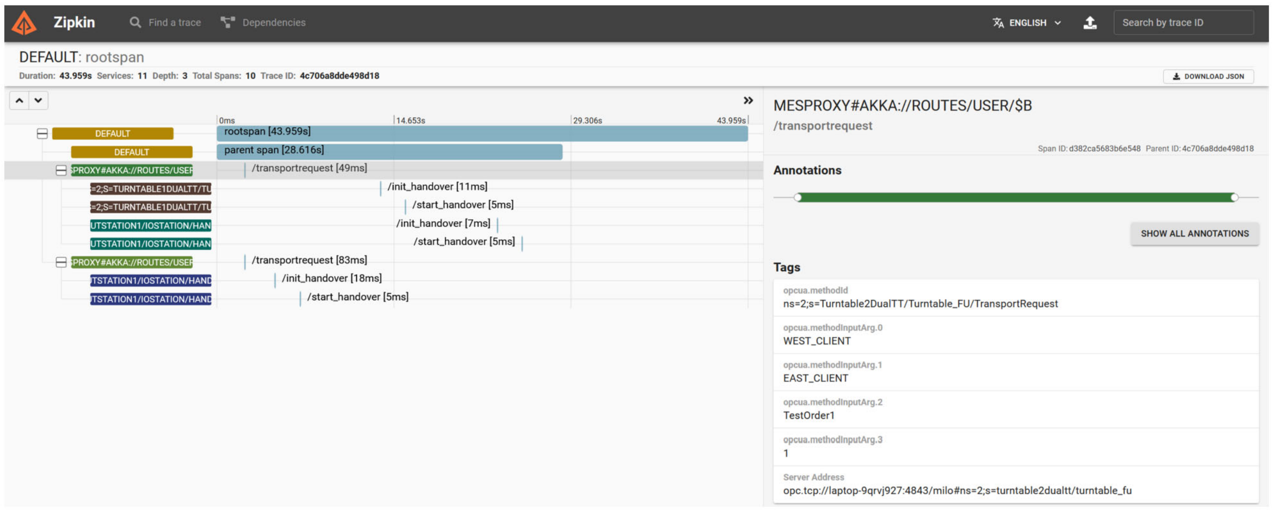
4. Evaluation use case

In the Factory in a Box scenario, the primary communication via OPC UA occurs between the MES and the machines (input station, output station, turntables, plotters) as well as between machines to coordinate the handover of pallets. In doing so, we can obtain a single trace from the MES' transport request(s) to the turntable(s) and the subsequent handover handshakes. Figure 2 provides a conceptual architecture overview of the involved components in the Factory in a Box for transporting a pallet from input station to Turntable 1, on to Turntable 2, a plotter, and ultimately reaching the output station. To this end, turntables offer an OPC UA server endpoint for receiving transport commands (i.e., method invocations), and then connect with respective OPC UA clients to the other machines to coordinate the pallet handover. A turntable also exposes its OPC UA server to other turntables to coordinate pallet handover between turntables. Note, that the input station and output station need not be controlled from the MES as they signal via their state (i.e., accessible as an OPC UA node) whether they are ready to provide a pallet, respectively, whether they can take on a pallet.

Figure 3 displays the resulting trace information as displayed in the Zipkin UI for an exemplary transport scenario where a pallet is moved from input station to output station via the turntables without, for the sake of clarity, involving a stop at a plotter. Note that a turntable needs to internally forward the trace correlation information obtained when receiving the transport request to the handshake component which coordinates the handover. This requires explicit extraction of the B3 header information from the OPC UA method call and thus is specific to the communication means used within the machines (here IEC 61499 for plotters, and Akka actor framework for turntables). The exact mechanism involved is outside the scope of this article. In FIAB, we limited the instrumentation to sending spans from the client-side as not all server-side OPC UA stacks involved in FIAB are yet instrumented (i.e., currently only Eclipse Milo, but not yet the open62514 server used in the Forte framework on the plotters and input and output stations).

There are several aspects visible in the trace of Fig. 3 (left). The root span is created in the MES (indicated by "DEFAULT" spans), where the transport system coordinator has separate OPC UA clients to connect to both involved turntables. A transport request is then dispatched to the two turntables at the same time. This is visible in the timestamps of the two "/transportrequest" spans that reside at nesting level two. Figure 3 (right) provides details of one such request in the span's tags, here specifically the method invocation parameters and OPC UA method node identifier.

The spans are listed in their dispatch order by timestamp. In this particular case, the MES happened to dispatch the transport request to Turntable 2 first, and immediately afterward to Turntable 1. Turntable 1 then proceeds to initiate the handshake to the input station (the two bottom spans). Only once Turntable 1 has completed loading the pallet and is in a ready state will Turntable 2 commence with the handshake to Turntable 1 (the two brown spans), and then the handshake to the output station. The time before and after the set of two handshake method calls is spent on turning and conveying (internal component behavior that involves no traceable OPC UA method calls). The attentive reader will notice that the spans' duration is extremely short. As we outlined in the previous section, the method calls are merely used as triggers to an action such as "initiate" or "start" a handshake and trigger a transport request. To obtain spans of longer duration, it is necessary to push the tracing instrumentation from the OPC UA stack up into the (sub) components that manage logic at a higher abstraction level, for example, the state machine that governs the handshake. There, a component could then start a span upon receiving an internal "start" re-



DEFAULT: rootspan

Duration: **43.959s** Services: **11** Depth: **3** Total Spans: **10** Trace ID: **4c706a8dde498d18**

```
                              0ms              14.653s              29.306s              43.959s
DEFAULT                       rootspan [43.959s]
  DEFAULT                     parent span [28.616s]
  PROXY#AKKA://ROUTES/USEF      /transportrequest [49ms]
    =2;S=TURNTABLE1DUALTT/TU                        /init_handover [11ms]
    =2;S=TURNTABLE1DUALTT/TU                          /start_handover [5ms]
    UTSTATION1/IOSTATION/HAN                        /init_handover [7ms]
    UTSTATION1/IOSTATION/HAN                          /start_handover [5ms]
  PROXY#AKKA://ROUTES/USEF      /transportrequest [83ms]
    TSTATION1/IOSTATION/HAND       /init_handover [18ms]
    TSTATION1/IOSTATION/HAND         /start_handover [5ms]
```

MESPROXY#AKKA://ROUTES/USER/$B

/transportrequest

Span ID: d382ca5683b6e548 Parent ID: 4c706a8dde498d18

Annotations

Tags

opcua.methodId	ns=2;s=Turntable2DualTT/Turntable_FU/TransportRequest
opcua.methodInputArg.0	WEST_CLIENT
opcua.methodInputArg.1	EAST_CLIENT
opcua.methodInputArg.2	TestOrder1
opcua.methodInputArg.3	1
Server Address	opc.tcp://laptop-9qrvj927:4843/milo#ns=2;s=turntable2dualtt/turntable_fu

Fig. 3. Example trace of a transport request from the MES to two turntables, as well as the handshake communication to an input station, among the turntables, and an output station. Span details of one request displayed on the right

quest, capture in span annotations the events triggered by receiving of state updates via monitored items, and eventually close the span upon transitioning into the handshake state machine's "Completed" state. Any OPC UA-level spans would then be captured as they currently are but only as sub spans of a higher-level handshake span. Similarly, the MES transport system coordinator would have to manage separate spans for each end-to-end transport request.

Nevertheless, already at the current level of trace instrumentation, the available trace information allows to inspect the timing of requests, here for example, between initiate and start, and analyze any deviations thereof over time, or the context of requests in case one of them fails.

5. Related work

A plethora of runtime monitoring approaches [8] focus on the monitoring of resources, e.g., for fault detection or resource usage [15], often supported by model-driven techniques to simplify instrumentation [6]. In this category, approaches that address distributed systems, mostly focus on aspects and events of the individual distributed components but not their interaction. Here passing of correlation metadata is less of a requirement, or implicitly provided in the identifier of the instrumented components [16].

As production systems become more flexible and need to support frequent reconfiguration on the move to lot-size one, per order, batch, or request tracing becomes important. To this end, metadata propagation based techniques such as Zipkin (originally developed by Twitter) are suitable candidates to support these existing approaches. Zipkin is not the only distributed tracing technology with multiple frameworks emerging over the past decade (e.g., Dapper [12] at Google, Jaeger [10] at Uber, or Canopy [3] at Facebook).

These tracing infrastructures are then the basis for sophisticated use cases such as anomaly detection [7] or attack path detection [14]. Additional use cases and their support by distributed tracing frameworks are discussed in depth by Sambasivan et al. [9]. As the number of different use cases extends to debugging, taint propagation, auditing, or provenance, a generic multipurpose context propagation mechanism becomes sensible [4].

Alternative approaches to metadata propagation-based monitoring include treating the distributed system as a black box and thus only derive insights from analyzing vast amounts of heterogeneous log entries (e.g., [2]) or schema-based approaches (e.g., [1]) that define a model for each component on how to interpret logged system calls. Such approaches, however, have significant limits in their ability to enable inspection of the causal relationships of requests (and their properties) for individual traces.

6. Conclusions and outlook

In this article, we motivated the potential for distributed tracing to better understand causality in highly distributed and dynamic CPPS. We introduced how to integrate Zipkin with OPC UA method calls and discussed that additional instrumentation is also useful when reading and writing to nodes via OPC UA Data Access. We also highlighted that due to the event-centric nature of CPPS, method call-based spans are not ideally suited to quickly and easily understand span structures. To this end, span generation based on component internal models such as state machines seems very promising. As we outlined in Sect. 5, Zipkin is just one applicable framework, with the OpenTracing APIs[6] aiming for interoperability. However, all similar approaches that have their origin in the web and cloud domain face similar challenges concerning passing correlation information and, most importantly, the event-centric nature of OPC UA communication.

Funding Note Open access funding provided by Johannes Kepler University Linz.

[6] https://opentracing.io/.

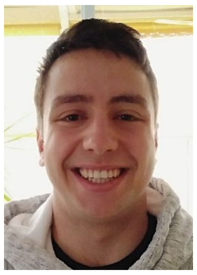
References

1. Barham, P., Donnelly, A., Isaacs, R., Mortier, R. (2004): Using magpie for request extraction and workload modelling. In 6th {USENIX} symposium on operating systems design and implementation ({OSDI}'4) (Vol. 4, pp. 18).
2. Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T. F. (2014): The mystery machine: end-to-end performance analysis of large-scale Internet services. In 11th {USENIX} symposium on operating systems design and implementation ({OSDI} 14) (pp. 217–231).
3. Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., et al. (2017): Canopy: an end-to-end performance tracing and analysis system. In Proceedings of the 26th symposium on operating systems principles (pp. 34–50).
4. Mace, J., Fonseca, R. (2018): Universal context propagation for distributed system instrumentation. In Proceedings of the thirteenth EuroSys conference (pp. 1–18).
5. Mahnke, W., Leitner, S. H., Damm, M. (2009): OPC unified architecture. Berlin: Springer.
6. Mazak, A., Lüder, A., Wolny, S., Wimmer, M., Winkler, D., Kirchheim, K., Rosendahl, R., Bayanifar, H. Biffl, S. (2018): Model-based generation of run-time data collection systems exploiting automationML. Automa, 66(10), 819–833. https://doi.org/10.1515/auto-2018-0022.
7. Nedelkoski, S., Cardoso, J., Kao, O. (2019): Anomaly detection and classification using distributed tracing and deep learning. In 2019 19th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID) (pp. 241–250). New York: IEEE.
8. Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grünbacher, P. (2017): A comparison framework for runtime monitoring approaches. J. Syst. Softw., 125, 309–321.
9. Sambasivan, R. R., Fonseca, R., Shafer, I., Ganger, G. R. (2014): So, you want to trace your distributed system? key design insights from years of practical experience. Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA. Tech Rep CMU-PDL 14.
10. Shkuro, Y. (2017): Evolving distributed tracing at uber engineering. Uber Engineering Blog.
11. Shkuro, Y. (2019): Mastering distributed tracing: analyzing performance in microservices and complex systems. Birmingham: Packt Publishing Ltd.
12. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure.
13. Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sunder, C., Valentini, A., Martel, A. (2008): Framework for distributed industrial automation and control (4diac). In 2008 6th IEEE international conference on industrial informatics (pp. 283–288).
14. Sun, X., Dai, J., Liu, P., Singhal, A., Yen, J. (2018): Using Bayesian networks for probabilistic identification of zero-day attack paths. IEEE Trans. Inf. Forensics Secur., 13(10), 2506–2521.
15. Van Hoorn, A., Waller, J., Hasselbring, W. (2012): Kieker: a framework for application performance monitoring and dynamic software analysis. In Proceedings of the 3rd ACM/SPEC international conference on performance engineering (pp. 247–248).
16. Vierhauser, M., Rabiser, R., Grünbacher, P., Seyerlehner, K., Wallner, S., Zeisel, H. (2016): Reminds: a flexible runtime monitoring framework for systems of systems. J. Syst. Softw., 112, 123–136.
17. Zimmermann, P., Axmann, E., Brandenbourger, B., Dorofeev, K., Mankowski, A., Zanini, P. (2019): Skill-based engineering and control on field-device-level with OPC UA. In 24th IEEE international conference on emerging technologies and factory automation, ETFA 2019, Zaragoza, Spain, September 10-13, 2019 (pp. 1101–1108). New York: IEEE. https://doi.org/10.1109/ETFA.2019.8869473.

Authors

Christoph Mayr-Dorn
is a senior researcher at the Institute for Software Systems Engineering at the Johannes Kepler University Linz, Austria. He holds a Ph.D. in Computer Science from the Technical University Vienna. His current research interests include software process monitoring and mining, change impact assessment, and software engineering of cyber-physical production systems.

Benno Pereszteghy
recently completed his Bachelors Programm at the Johannes Kepler University Linz, Austria.



Jan Holzweber
is a Bachelor Student at the Johannes Kepler University Linz, Austria and programmer at the *Pro*^{2}*Future* competence center.

Michael Mayrhofer
is Area Manager for "Cognitive Robotics and Shopfloors" at the *Pro*^{2}*Future* competence center and completing his PhD at the Johannes Kepler University Linz, Austria.