Throttled Service Calls in OPC UA

Ahmed Ismail and Wolfgang Kastner
Institute of Computer Aided Automation - Technische Universität Wien
Vienna, Austria
Email: {aismail, k}@auto.tuwien.ac.at

Abstract—The Open Platform Communications Unified Architecture (OPC UA) standard specifies the use of a push-based communication model for service calls in its client-server architecture. This paper clarifies how this model leaves OPC UA Servers vulnerable to request overload. A solution based on rate throttling is demonstrated to counter this vulnerability. The requirements of the resulting service are quantified and an open-source prototypical implementation is built using the Apache ZooKeeper and open62541 libraries.

I. Introduction

A widely accepted standard for machine-to-machine (M2M) communication in the manufacturing industry is the Open Platform Communication (OPC) Unified Architecture (UA). OPC UA is a set of specifications for client-server data exchange. It primarily defines a Service Oriented Architecture (SOA) and an information model. Thus, it provides a standardised set of pre-defined services, interfaces, protocols, and formats for data representation and structuring [1].

Communication between clients and servers in OPC UA is in the form of service calls. Effectively, these are remote procedure calls (RPC). This implies that OPC UA predominantly operates using a client-side push-based communication model. The problem with this model is that it is possible for a single OPC UA Server to receive too many concurrent requests. In trying to process these requests, the server may exhaust or over-extend its available resources. Thus, the server may enter a failed or degraded state causing clients to experience high latencies, request time-outs, or service unavailability [2]. The degraded operation or unavailability of a component in an online manufacturing system is highly undesirable. The National Institute of Standards and Technology (NIST), for example, recorded an incident where a hanged control system in a wafer fabrication plant caused a financial loss worth US $ 50 000 [3].

It can be argued that the use of OPC UA Server Redundancy Groups and ServiceLevel indicators may alleviate the symptoms of server overload or prevent them from occurring. The former, as the name implies, involves having sets of redundant servers with access to the same underlying resources and a synchronised information model. The latter, on the other hand, is a byte variable included in every OPC UA Server address space. It can have a value of 0, 1, 2-199, or 200-255 signalling that the Server is in a Maintenance, NoData (failed), Degraded, or Healthy state, respectively. According to the specifications, clients connected to a Degraded Server should not expect reliable services. Consequently, the client is permitted to switch to another healthy Server, if one is available. The client may also connect to multiple degraded servers to maximize its access to the underlying devices and their data. If connecting to a healthy server, a client is expected to select the one with the highest value. The sum of these tools therefore amounts to capacity planning and a simple load balancing strategy [4].

However, since capacity planning and load balancing do not alter the client-side push-based communication model of OPC UA, servers continue to be vulnerable to overload. In fact, as clients are permitted to switch from degraded to healthy servers, the situation may worsen as failures cascade across an impacted redundancy set and possibly cause the entire service to fail.

Other possible solutions limit the number of requests that are received and/or processed by an OPC UA Server, thus implying rate throttling and load shedding, respectively. Both of these strategies involve the use of queues. Rate throttling would use a resourceful mediator to shield an OPC UA Server from traffic bursts. In contrast, load shedding operates on the premise that rejecting a service call uses significantly less resources than processing it. By limiting the number of requests being concurrently processed and rejecting the rest, servers are believed to reduce their chances of failing [5]. While both may be viable solutions to the problem, neither approach appears to have been investigated within this context. Thus, this paper takes the first step by examining the use of a rate throttling mediator to combat server overload in OPC UA.

Section II begins by discerning the requirements for the queuing service. Section III then demonstrates how features of the coordination tool, Apache ZooKeeper, can be used to meet these requirements. Section IV proceeds to describe an open-source prototype developed to evaluate a suitable architecture, data model, and communication flow for rate throttling based on ZooKeeper. Section V discusses concerns surrounding the presented system and future work. Finally, Section VI concludes the paper.

II. Requirements

This section discerns the requirements necessary of a mediator for the queuing of OPC UA service calls. These requirements are summarized in Table I.

TABLE I
Criteria for a task queuing service.

Parameter	Requirement
Scalability	The system should be able to scale to support 1000's to 100 000's of connected OPC UA Clients and Servers.
Consistency Guarantees	The service must reflect a consistent state.
Recoverability	The service should be able to recover from system and network problems.
Security	The system must, at least, provide the same security features as OPC UA.
Client-Push/Server-Pull Communication	OPC UA Clients must push to the queueing service, while Servers must pull tasks off their respective queues when they wish to receive the data.
At Least Once Semantics	Every service call should be delivered to a server at least once.
Supportability	The service must be well-documented, actively developed, and have a healthy support community.

To begin, the queueing service is expected to be highly scalable. It should be able to support thousands to hundreds of thousands of concurrent connections from both OPC UA servers and clients. Scalability should, in this case, be horizontal due to the associated benefits [6]. This means that the tool should be able to operate as a distributed system.

As for the service calls, submissions should be committed in a transactional and strongly consistent manner. According to [7], this implies four properties:

- Atomicity: An operation either succeeds or fails; inconsistent states are not permitted.
- Consistency: "Committed transactions are visible to all future transactions" [7]. This means that all redundant participants in the queueing service apply transactions in the same order, thereby preserving the uniformity of the service state [8].
- Isolation: Uncommitted transactions are not visible to future transactions.
- Durability: Transaction commits are permanent.

These are important properties for task queuing in safety-critical environments. Partial or non-permanent commits, fuzzy reads, and dirty reads and/or writes imply that an online system may operate out of specification. Given the tight and complex coupling between software and physical processes in cyber-physical production systems (CPPS), the outcome may manifest as undesirable physical events.

A third point to address is that of availability and fault-tolerance. In an ideal system, every submitted service call should be successfully queued and subsequently made accessible to their respective OPC UA Servers. However, system and network problems are to be expected. Measures should therefore be included to tolerate crashed node faults. These would allow connected OPC UA Clients and Servers to continue using the queueing service in cases of partial system failures. The service should also have protections in place against network partitioning. If the system does not protect against network partitions, split-brain behaviour may manifest. This is when fractions of the service may progress the state of the system independently and in an inconsistent manner which may have an undesirable effect on the manufacturing environment.

Naturally, clients must be able to submit tasks for execution with associated metadata. Likewise, servers must be able to retrieve them. At-least-once message delivery semantics is necessary to guarantee that each service call is consumed by a server. Messages should also be delivered to servers using pull-based mechanisms to mitigate the previously discussed server overload scenarios.

Moreover, the system should match, if not improve upon, the security features inherent to OPC UA. The system should therefore incorporate measures to ensure that the submission, retrieval, and modification of queued tasks are only done by authorized clients. This amounts to the inclusion of authentication and encrypted communication features.

The last point to discuss is that of supportability. The tool should have strong community and developer support available. It should be well-documented and actively developed to ensure a long and stable lifetime for the service.

With these requirements in place, the coming section will discuss a suitable tool for the implementation of the queueing service.

III. Queuing Service

Several of the features identified in the previous section, such as scalability, high availability, and security, are generic enough to allow for the applicability of a broad selection of tools for the queuing service. One of the more difficult features required of the queuing service is that of strong crash fault tolerant consistency guarantees. Algorithms for fault tolerant consensus and high availability in asynchronous environments include Paxos, raft, and the Viewstamped Replication protocol [9, 10]. In [11], however, it is shown that the most widely adopted tool for consensus-dependent applications is ZooKeeper. This is an application which operates using a Paxos-inspired variant known as the ZooKeeper Atomic Broadcast (Zab) protocol. In this section, we describe the relevant features and strengths of ZooKeeper that demonstrate its viability as a queuing service for OPC UA. A summary of these features may be found in Table II. Supportability is considered a met requirement due to ZooKeeper's wide adoption, detailed documentation[1], and frequent updates[2]. For a more complete description of ZooKeeper, please refer to [12], [13], and [14].

TABLE II
Features of ZooKeeper and the criteria for a task queueing service.

Parameter	Features
Scalability	Quorum mode, observers, dynamic scaling.
Consistency Guarantees	Zab protocol, majority quorum in an odd numbered ensemble.
Recoverability	Zab protocol, transaction logs, snapshots.
Security	Client-server and server-server mutual authentication, client-server SSL, znode-level access control.
Client-Push/Server-Pull Communication	Write/read operations and watches.
At Least Once Semantics	Watches, persistent znodes, and read operations.
Supportability	Mature, widely adopted product [11].

A. Servers, Quorums, and the Zab Protocol

ZooKeeper is a client-server application for distributed consensus. Clients can interact with the service using a client library. The service can run in standalone or quorum mode. The former requires only one server. Thus, no replication of ZooKeeper's state occurs. The latter, on the other hand, uses a group of servers, termed the ensemble, which replicate ZooKeeper's state and serve client requests.

In quorum mode, a server can be a leader, follower, or observer. Clients can establish a mutually-authenticated and SSL-encrypted TCP session with any of the available servers (unauthenticated and unencrypted sessions are also possible). Clients can submit read or state-changing requests that are executed first-in-first-out (FIFO). Requests are atomic and do not permit partial results. Read requests are executed locally by the connected server while state-changing requests are forwarded to the leader. The leader is an elected server that is responsible for executing and ordering requested state changes using the Zab protocol. This is a two-phase commit protocol that operates as follows:

1) A requested state change is transformed by the leader into a transaction that includes the steps needed to atomically apply the state change.
2) The transaction is transmitted by the leader to its followers as a proposal.
3) Each follower checks that the proposal is from its current active leader and that it conforms with the current order of acknowledged and committed transactions.
4) If the proposal is found to be compliant, the followers accept the proposal and respond to the leader with an acknowledgement.
5) The minimum number of servers that need to respond to a proposal with an acknowledgement is referred to as the quorum. Once the leader receives enough acknowledgements, it sends out a commit message to the followers.

By following these steps, the protocol ensures that a state change is properly stored before it is committed. It also guarantees that transactions are ordered, consistent, and durable, despite possible crash faults. It's also worth mentioning that changes are also permanent as ZooKeeper does not support rollbacks.

To protect against split-brain behaviour in a ZooKeeper system, the total number of servers in the ensemble should be odd and the quorum should be a majority of the available servers. Therefore, if a network partition does occur, only one partition may have the number of servers needed to progress ZooKeeper's state.

As for scalability, ZooKeeper achieves this through the use of observer servers and dynamic scaling. Observers are servers that replicate the state of ZooKeeper. They are used to scale the system without impacting the performance of state-changing requests as they do not participate in the voting process. Dynamic scaling, on the other hand, allows a ZooKeeper service to add new servers while the system is online.

So far, this subsection has addressed the scalability, consistency guarantees, security, and client-push/server-pull communication requirements in Table I. The next subsection will discuss ZooKeeper's data structure to show how at-least-once message delivery may be achieved.

B. Data Structure

ZooKeeper stores data in units called znodes that are organized into a hierarchical tree. Znodes can be ephemeral or persistent. Ephemeral znodes only exist as long as the client that created it has a valid session with ZooKeeper. Ephemeral znodes are therefore not permitted to have children. Persistent znodes, on the other hand, are only removed if an authorized client calls for their deletion. Persistent znodes are therefore used to grow the data tree. Both types of znodes can be sequential. This implies the assignment of a unique and sequentially incremented integer to a znode's name by ZooKeeper. To secure the data tree, access rights are assigned to znodes each time a znode is created.

Clients can register for notifications on state changes in any znode by setting a watch. A watch can be set for the creation or deletion of a znode, or for changes in a znode's data or its children. If a watch is triggered, a single notification is sent to the client and the watch is removed. The watch must reset to continue monitoring the znode. Watches are set using read operations to avoid missing state changes in the time between when the notification is triggered and the watch is reset. Watches persist across servers, but, similar to ephemeral znodes, are removed if the client's session expires.

[1]https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index
[2]https://zookeeper.apache.org/releases.html

Thus, as long as permanent znodes with proper access rights are used for the submission of service calls, OPC UA Servers should always be able to retrieve submitted tasks at least once. Therefore, this achieves the required at-least-once message delivery semantics in Table I. For recoverability, the next subsection presents the data persistence features of ZooKeeper.

C. Data Persistence

ZooKeeper uses transaction logs and snapshots for data persistence on local storage. These are both files on the local filesystem. Proposals are appended to a transaction log before they are accepted to persist ordered transactions to disk. A snapshot, on the other hand, is a copy of ZooKeeper's data tree serialized to file. ZooKeeper snapshots are fuzzy as requests continue to be processed while a snapshot is being taken. The transaction log must be replayed when loading a snapshot to retrieve the true state of a tree at a specific point in time. Thus, snapshots and transaction logs may be used for recovery purposes in case of node crashes.

With all of the requirements of Table I met, the next section details the design and implementation of a queuing service using ZooKeeper.

IV. Service Call Throttling

This section is concerned with using the mechanisms available to ZooKeeper to implement a queuing service. As such, it specifies the requirements and implementation details of an integrated OPC UA and ZooKeeper solution for mediating OPC UA service calls.

A. Demands

There are several requirements for the operation of the queue. First, the system should allow OPC UA Servers to register and initialize a queue. Furthermore, OPC UA Clients and Servers must be able to assign and retrieve tasks to and from their respective queues. Also, the order in which tasks are processed may be important. As such, the system should give sufficient support for ordered execution. The system should also have mechanisms that allow OPC UA Clients to detect if a Server has crashed or disconnected. This may be useful, for example, when an OPC UA Client can communicate directly with an OPC UA Server, but the Server is unable to communicate with and retrieve assigned tasks from ZooKeeper.

OPC UA Clients may also need to be able to circumvent the queue if necessary. For example, in a safety-critical environment, certain tasks may need to be processed by a server immediately, regardless of the length of the queue. Thus, it may be necessary to keep an OPC UA-native back-channel open to allow Clients to invoke service calls on Servers directly.

Lastly, a typical pattern for message oriented middleware (MOM) based message queuing involves publishing the response message on the same platform [2]. This is

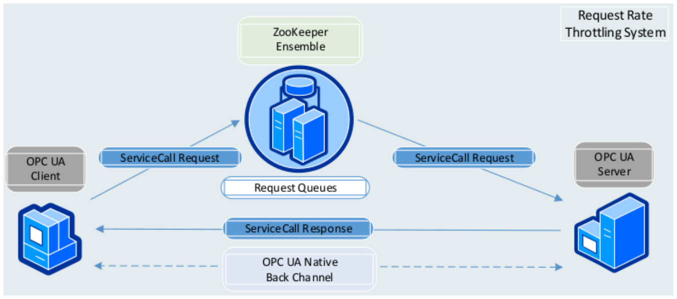

Fig. 1. The integrated system architecture.

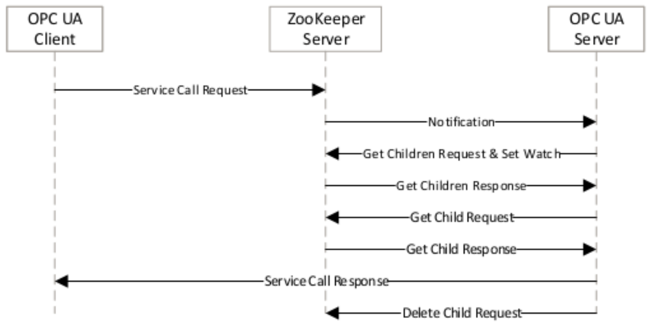

Fig. 2. Changes to the client-server service call communication flow.

not necessary for the queueing service. This is because the service's purpose is to shield OPC UA Servers from excessive concurrent service calls by Clients, and not vice versa. Hence, there is no immediate need to queue Server responses as well.

With these demands in place, the next subsection will describe the implementation of a ZooKeeper queuing service. This includes the data structure employed and the expected communication flow between an OPC UA Client, ZooKeeper server, and OPC UA Server.

B. Implementation

A prototypical implementation of the queueing service is built using an open source C99 implementation of OPC UA, open62541 [15], and the ZooKeeper library [14]. This subsection presents the data structure and the communication flow used for the the implementation, which itself is also open source[3].

The data structure used by the queueing service builds upon previous work employed in [13]. In [13], ZooKeeper is used for the coordination of redundant OPC UA Servers. The main feature adopted is the use of a Global Unique Identifier (GUID) for each set of redundant OPC UA Servers. In the queueing service, this allows clients to set watches in a more specific manner that is expected to reduce the overall frequency of notifications received. The resulting data structure for service registration, queue management, and crash detection would therefore be as shown in Table III.

[3]https://github.com/AGIsmail/UaRateThrottling

TABLE III
The ZooKeeper Data Model for OPC UA Service Call Queuing

ZooKeeper Path	Type of zNode	Explanation
/Servers	Persistent	The root folder for Zookeeper integrated OPC UA (zkUA) Servers.
/Servers/{GroupGUID}	Persistent	A GUID unique to each OPC UA Server redundancy set.
/Servers/{GroupGUID}/Active	Persistent	Path to registered zkUA Servers that are in a functional state and connected to a downstream device.
/Servers/{GroupGUID}/Active/{ServerUri}	Ephemeral	Each active server registers itself using an ephemeral znode.
/Servers/{GroupGUID}/Queue/	Persistent	Path to the queues of registered zkUA Servers that are in a functional state and connected to a downstream device.
/Servers/{GroupGUID}/Queue/{ServerUri}	Persistent	Every OPC UA Server creates a persistent znode named after its server URI under the Queue path to accept task assignments.
/Servers/{GroupGUID}/Queue/{ServerUri}/Tasks-**********	Persistent-Sequential	Clients assign tasks to a Server by creating a znode with the service call and the needed arguments under the correct Server's Tasks path. Each task is a sequential znode for the synchronous execution of calls.

First, the root path for the queuing service on ZooKeeper is the /Servers znode. Each set of redundant OPC UA Servers is assigned its own unique path under the /Servers znode using its GUID. The resulting path is therefore /Servers/GroupGUID. Any active server in the redundancy set then registers itself under the /Servers/GroupGUID/Active znode using its ServerUri or ServerId. A ServerUri or ServerId is used to identify a specific server out of a redundancy set in the case of non-transparent or transparent server redundancy, respectively.

Once an OPC UA Server registers itself as an active server, it initializes a queue using its ServerUri/ServerId under the Servers/{GroupGUID}/Queue/ znode. OPC UA Clients can submit tasks to a Server's queue as permanent and sequential znodes. Thus, after the queue has been initialised, the Server sets a watch on its tasks queue using a read operation to monitor for new tasks. If tasks have already been queued between the time that the queue is initialized and read, they are retrieved, processed, and deleted from ZooKeeper. As a watch is set on the queue by the Server, any future tasks added by Clients trigger a single notification with the creation of a task znode. No further notifications are sent until the Server retrieves the tasks list and re-sets the watch. The full communication flow between an OPC UA Client, ZooKeeper Server, and OPC UA Server is demonstrated in Fig. 1-2.

V. Discussion

The previous section presented the data structure and process flow for the queuing and processing of OPC UA service calls. An immediate concern revolves around the possibility of service calls being processed more than once or not at all. The former may occur if the queue is retrieved more than once before the OPC UA Server has had a chance to process and delete all of the tasks retrieved in a previous run from ZooKeeper. The prototype currently prevents this from occurring Once a notification is triggered, the watcher function synchronously retrieves the tasks list, re-sets the watch, and processes each task before returning. The watcher function is not triggered again until it has completed, and therefore deleted the tasks on ZooKeeper and returned. This is the desired order of events as it is a demand of section IV-A that tasks be executed in order.

If, however, the ordered execution of service calls is not important, then the watcher function may asynchronously execute its tasks and return early. This may allow tasks to be executed more than once. Recall, however, that tasks are submitted to ZooKeeper as sequential znodes. Preventing the multiple executions of a service call could therefore be as simple as having each thread processing a task list only execute znodes with an ID higher than the last ID of its predecessor and lower than its own last task's ID. Alternatively, an internal queue, e.g., using a hashtable, could be used to store the retrieved tasks and their status. Multiple threads can then contend over the available tasks and use locks to relieve the possibility of duplicate executions.

Another issue to be addressed is the possibility that service calls time out while in the queue. However, the OPC UA specifications do not currently specify a value for message time-outs. This value is purposely left open for developers to set. Needless time-outs can therefore be avoided through the selection of a reasonable value that reflects the context of execution.

Concerning Clients' ability to circumvent the queue, this should be done with the utmost care. The use of OPC UA's native communication flow for service calls effectively pushes requests to the front of the queue. This may overload a server and cause it to enter a degraded or failed state. It may also alter the states previously used to submit the currently queued requests and may result in their unsafe execution. Out-of-queue service calls should therefore be done with caution.

Finally, in [11], it is indicated that distributed queues is one of the least used applications for ZooKeeper. The explanation offered is that consensus may have a detrimental effect on the performance of large queues. Apache Curator's documentation[4] recommends against the use of ZooKeeper for queues because of an anecdotal report. The report mentions ZooKeeper's transport limitations, slow start up times and other complexities introduced by having large queues, and other factors. However, the points listed are all self-admittedly born of casual observation in a global and multi-tenant architecture. Thus, these remarks are not based on rigorous scientific analysis, are not quantified, and are put forth by a single source. It is also stated as being the result of "[developers abusing] the queues"[5]. Given the difference in context, results may differ. This paper does not currently pursue an evaluation to discern these limits which may include the size of the ZooKeeper ensemble, number of active OPC UA clients and servers, znode size and distribution per size, queue lengths, ZooKeeper settings (e.g., forceSync), and hardware performance (e.g., storage I/O) [16]. This, in fact, is a problem that may garner different results between deployments. Instead, the prototype developed for this paper is open-sourced and it should be possible for future work to design appropriate architectures, quantify normal limits of operation, and address contingencies for the scalable execution of the service and dependent systems once feedback is available from early adopters.

Further future work may also investigate the use of load shedding for server overloads as was mentioned in Section I. A comparison of server-side load-shedding and task queueing on ZooKeeper would be immensely useful. A possible outcome of this analysis, for example, may shed light on the engineering cost involved in determining which service calls can be dropped versus the cost of ensuring safe out-of-queue service calls.

VI. Conclusion

This paper describes the possible vulnerability of OPC UA Servers to resource exhaustion due to high rates of concurrent service calls by OPC UA Clients. A queuing service for service call rate throttling is identified as a possible solution for mitigating server overload. The requirements for this service are determined and shown to be achievable using Apache ZooKeeper. A data structure and queuing protocol is designed and demonstrated using a prototypical implementation based on the open62541 and ZooKeeper libraries.

While certain facets of the design necessitate care in use and administration, the system meets all of the demands determined for the reliable queuing and execution of service calls. Future work is expected to focus on comparisons with alternative solutions, as well as the determination of appropriate deployment architectures and best-practices that account for identified contingencies.

[4]https://curator.apache.org/curator-recipes/distributed-queue.html

[5]https://cwiki.apache.org/confluence/display/CURATOR/TN4

Acknowledgment

This paper is supported by TU Wien research funds as part of the Doctoral College CPPS.

References

[1] W. Mahnke, Stefan-Helmut Leitner, et al. OPC Unified Architecture. Springer Berlin Heidelberg, 2009.

[2] Gregor Hohpe and Bobby Woolf. Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley, 2004.

[3] Keith Stouffer, Victoria Pillitteri, et al. NIST Special Publication 800-82 Revision 2: Guide to Industrial Control Systems (ICS) Security. Tech. rep. May 2015.

[4] Part 4: Services. R1.03. OPC Foundation. July 2015.

[5] Evan Jones. Preventing server overload: limit requests being processed. url: http://www.evanjones.ca/prevent-server-overload.html.

[6] Dilpreet Singh and Chandan K. Reddy. "A survey on platforms for big data analytics". In: Journal of Big Data 2.1 (2015).

[7] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: Acm Sigact News 33.2 (2002).

[8] Michael J. Fischer, Nancy A. Lynch, et al. "Impossibility of distributed consensus with one faulty process". In: ACM Press, 1983.

[9] Lewis Tseng. "Recent Results on Fault-Tolerant Consensus in Message-Passing Networks". In: 2016 SIROCCO. Ed. by Jukka Suomela.

[10] Robbert van Renesse, Nicolas Schiper, et al. "Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab". In: IEEE Transactions on Dependable and Secure Computing 12.4 (July 2015).

[11] A. Ailijiang, A. Charapko, et al. "Consensus in the Cloud: Paxos Systems Demystified". In: 2016 ICCCN. Aug. 2016.

[12] Flavio Junqueira and Benjamin Reed. ZooKeeper. O'Reilly Media, 2013.

[13] Ahmed Ismail and Wolfgang Kastner. "Coordinating Redundant OPC UA Servers". In: 2017 IEEE ETFA.

[14] Patrick Hunt, Mahadev Konar, et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: USENIX Annual Technical Conference. Vol. 8. 2010.

[15] F. Palm, S. Grüner, et al. "Open source as enabler for OPC UA in industrial automation". In: 2015 IEEE ETFA.

[16] Sanket Chintapalli, Derek Dagit, et al. "PaceMaker: When ZooKeeper Arteries Get Clogged in Storm Clusters". In: 2016 IEEE CLOUD. 2016.