© © 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Coordinating Redundant OPC UA Servers

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

***Abstract*—Coordination is an important aspect of large scale distributed systems. The Open Platform Communications Unified Architecture (OPC UA) standard series presents an important challenge for coordination in its specifications for server redundancy. This paper discerns the coordination needs of OPC UA server redundancy and presents a solution based on the integration of OPC UA and ZooKeeper. A detailed description of the architecture, data model, and components of the resulting system is given. This is accompanied by a discussion of an implementation based on the open source ZooKeeper and open62541 libraries. The resulting system is shown to be capable of meeting the coordination needs of OPC UA redundancy.**

I. Introduction

The increasing digitization of the manufacturing ecosystem is met with calls for the adoption of protocols and applications from the Information Technology (IT) domain to Operations Technology (OT). This is in order to leverage the benefits of IT in various fields and enhance the competitiveness of manufacturing enterprises. Academia has led such efforts over the years as is demonstrated by research in applying service oriented architectures (SOA), web services (WS), message oriented middleware (MOM), and other technologies to enhance the features of manufacturing systems [1].

These efforts are mirrored in industry by standardisation bodies and vendors alike. The OPC Foundation bases its new iteration of the OPC standard for machine to machine (M2M) communication, OPC UA, on a SOA and includes elements of the WS specification in its communication stack [2]. Yet, further enhancements to the OPC UA standard using solutions from IT are still possible. This is because OPC UA has gaps that are left open for vendor-specific implementations.

For example, a SCADA system based on OPC UA is typically a large distributed system. It is “a system comprised of multiple software components running independently and concurrently across multiple physical machines” [3]. As noted in [4], large distributed systems need several forms of coordination. This includes requirements for configuration, replication, synchronisation, group membership, discovery, leader election, and barrier synchronisation (resource fencing and locks) [4, 5]. For instance, OPC UA Servers running in Warm redundancy failover mode need mechanisms for address space synchronisation and to ensure that only one server in the redundancy set connects to a downstream device at a time [6].

Since the need for coordination mechanisms is common across many distributed applications, Yahoo! created and open-sourced a generalised service for coordination, named ZooKeeper. This service, with its uncomplicated filesystem-like API and strong guarantees for consistency, ordering, and durability has since become the de facto solution for distributed coordination [3].

This paper concerns itself with demonstrating the coordination of OPC UA server redundancy sets using the ZooKeeper service. As such, Section II begins by providing a primer on ZooKeeper. Section III then provides a brief overview of OPC UA and an in-depth description of redundancy in OPC UA. Section IV details the architecture, data model, and components used for coordinating redundancy sets with ZooKeeper. Section V addresses the implementation and Section VI discusses caveats and other points related to the system. Finally, Section VII concludes the paper.

II. ZooKeeper Overview

This section presents an overview of ZooKeeper by discussing its data structure, architecture, sessions, ZooKeeper Atomic Broadcast (Zab) protocol, local storage, failure resolution, and security [3].

A. Servers

The ZooKeeper service operates using a server-client architecture. Applications use a client library to interact with the Zookeeper servers. The servers can run in either standalone or quorum mode. The former is a single server and no replication of the ZooKeeper’s state occurs. In quorum mode, a group of servers, termed the ensemble, work together to replicate the ZooKeeper state and serve client requests. A quorum is the smallest number of servers out of the ensemble that is needed to allow ZooKeeper to work.

Within an ensemble, a ZooKeeper server can be either a leader, a follower, or an observer. The leader is an elected position that manages all requests for state changes, including the ordering of changes. The leader transmits state changes as proposals that are voted on by the followers to ensure the replication of state changes across the quorum. Observers do not vote on state changes and only replicate committed updates. Observers are typically used to scale the system.

B. Sessions

Clients connect to any single server using a TCP session. Clients send heartbeats to keep sessions alive. Only a server can declare a session as expired. If, however, a client does not hear from its server for a certain amount of time, the session may move to another server. The server it connects

to is selected at random as a form of simple load balancing. Within a single session, requests are executed first-in-first-out (FIFO). However, FIFO guarantees do not apply to concurrent and consecutive sessions.

C. Data Structure

ZooKeeper uses a hierarchical tree of data units, termed znodes, for its data structure. Znodes can be persistent or ephemeral. A persistent znode can only be removed through a call for deletion while an ephemeral znode is also removed if the session of the client that created it expires. Because ephemeral znodes are session-dependent they are not permitted to have child znodes. Both persistent and ephemeral znodes can also be sequential. Sequential znodes are assigned unique sequentially incremented integers. ZooKeeper supports the implementation of a quota on the number of znodes, called the count, and the size of data, called the bytes, that can be stored. Exceeding the set quota only causes a warning to be logged and does not interrupt the operation of the system.

Every znode is given a version number that is incremented with every change to its data. This allows for the conditional execution of certain operations, such as deletion and data setting operations. However, as a znode's version number is reset if it is deleted and re-created, conditional execution is not a fool-proof measure.

D. Watches and Notifications

Due to the performance penalty incurred by polling mechanisms, ZooKeeper favours a method based on notifications. Clients register for a notification by setting a watch on ZooKeeper. A watch is removed if the notification is triggered. To continue monitoring the znode, the client must therefore reset the watch. To avoid missing changes between receiving a notification and resetting the watch, watches are set using operations that read the znode's state.

Watches can be set to monitor for changes to a znode's data, its children znodes, or its creation or deletion. They are persistent across servers and can only be removed by being triggered or if the creating client's session expires.

Watches should be applied conservatively as they consume about 250-300 bytes of memory per watch, and the number of notifications sent for each watch is equal to the number of watches set for that znode. This rule of proportionality may result in undesirable traffic spikes on the network.

A final consideration related to watches relates to the 'Exists' watch which is set to monitor for the creation of a znode. As a node's creation may be missed between the time that a client disconnects and reconnects, it should only be used for long-lasting znodes.

E. Requests and the Zab Protocol

Requests can be read requests or state changing requests. Both are atomic and either succeed or fail; no partial results are permitted. Reads are executed locally by ZooKeeper servers, while the writes are forwarded to the leader. The request is typically initiated by the client. The leader transforms the request into a transaction that describes the steps to be applied atomically and results in a state change. These transactions are committed using the Zab protocol.

The Zab protocol requires that the leader sends the transaction to its followers as a proposal. The followers respond to the leader with an acknowledgement if they accept the proposal. Once the leader receives a majority of acknowledgements from the quorum it transmits a commit message to the followers and an inform message to the observers. It is important to note that transactions are both idempotent and permanent. ZooKeeper does not support rollbacks.

Transactions generated by the leader are assigned a ZooKeeper transaction ID (zxid). Each zxid is a 64-bit integer used to ensure that transactions are applied in the order established by the leader, amongst other things. As mentioned earlier, the zxid may be used for conditional execution.

F. Local Storage

ZooKeeper uses a pre-allocated transaction log file to persist ordered transactions on local storage. The transaction log is appended with proposals before they are accepted. ZooKeeper also offers snapshots, which are complete copies of the data tree serialized to file. Processes continue to execute during the snapshot taking process resulting in *fuzzy* snapshots that do not represent the true state of a tree at any specific point in time. However, this shortcoming may be remedied by replaying the transaction logs over the snapshot. Together, logfiles and snapshots may be used to recreate a server's state for later review or recovery.

G. Failures

Failures in ZooKeeper may occur in the service, network, or application and may be recoverable or unrecoverable failures.

Recoverable failures, such as a network hiccup, are normal events and applications are written to continue running in spite of them. In the case of the leader, all actions should be suspended while in a disconnected state as no updates are received during this time. Clients, on the other hand, lose all of their submitted requests when they disconnect and need to resubmit them once they reconnect.

Unrecoverable failures are typically caused by expired sessions or an authenticated session no longer being able to authenticate itself. Unrecoverable failures, should be handled by exiting the application. Once the application starts again, it may resynchronise with ZooKeeper. This avoids the possibility of undesirable manipulations of data in multi-threaded applications that automate recovery.

H. Security

The security aspects of ZooKeeper include:

1) access control lists (ACL): access rights are normally handled by the developer as they are set each time a znode is created. Access rights are not inherited by child znodes from their parent.
2) encrypted communication: client-server communication may be encrypted if the server has Netty and SSL

support while quorum communication does not currently support SSL [7].

III. OPC UA

This section provides an overview of the OPC UA standard in general and of redundancy in specific [2, 6, 8, 9, 10].

A. Overview

In its simplest terms, the OPC UA standard is composed of a set of specifications for the definition of data transfer software interfaces in a client-server architecture. Its two main technical properties are its service oriented architecture and information model.

The former provides a fixed set of services to establish communications between servers and clients and allows clients to interact with the application and information model. As such, the defined service groups are Discovery, SecureChannel, Session, NodeManagement, View, Query, Attribute, Method, Subscription, and MonitoredItem.

The latter, the information model, provides a flexible address space for modelling, exposing, and consuming networks of data and metadata of varying degrees of complexity. To do so, the OPC UA standard bases its model on two elementary units, *Nodes* and *References*.

Nodes are the simplest units of information and can be of various *NodeClasses* (types) that are predefined by the standard with specific *Attributes* (properties). The available NodeClasses are "*Object*, *ObjectType*, *Variable*, *VariableType*, *DataType*, *ReferenceType*, *Method*, and *View*" [9]. An Object node represents an abstract or physical component in the modelled system. A Variable node is used to store a value. A Method node is used to expose a function so that it may be called remotely. A View node is used to specify a subset of an address space. Last of all, the ObjectType, VariableType, DataType, and ReferenceType nodes, as their names imply, are used for the specification of Object, Variable, data, and Reference types, respectively.

Every node in the address space is identified and addressed using a unique *NodeId*. In addition to the canonical NodeId, a node may also have alternative NodeIds. Each NodeId is composed of an identifier and a *Namespace*. The namespace is used to allow naming authorities, such as vendors and organizations, to define unique NodeIds.

References, on the other hand, are used to connect nodes for organizational or filtration purposes. Each reference therefore has a source and destination node, a direction, and a *ReferenceType* used to indicate the properties and meaning of a reference.

B. OPC UA Server Redundancy

OPC UA supports the redundancy of both clients and servers to allow for availability, fault tolerance, and load balancing in different deployments. In OPC UA, this is achieved by allowing for duplicate instances of clients and servers. Special services, mechanisms, nodes, and client/server profiles are included in the specifications to support the various possible redundancy scenarios. These scenarios translate to a variety of failover modes that are available for OPC UA Server redundancy. A specific node is included in the address space, the *ServerRedundancy* node, to advertise the failover mode supported by an OPC UA Server. The different types and their requirements are detailed in the rest of this subsection.

1) Transparent Redundancy: Redundant servers operating in transparent redundancy (TR) mode all run using an identical server Uniform Resource Identifier (URI) and endpoint Uniform Resource Locator (URL). The servers therefore all appear as one server to connected clients. To allow a connected client to pinpoint the exact source of its data in a redundant server set, each server offers a unique ServerId. Information synchronisation between servers is the responsibility of the servers such that no actions are required by the client when a failover occurs.

2) Non-Transparent Redundancy: In non-transparent redundancy (NTR) mode, clients are expected to participate in the failover. This typically involves selecting a server to failover to and moving session-relevant information to the new server. For it to be able to do so, a server in NTR provides its clients with information on its failover mode and the other servers in its redundancy set. The different modes available in NTR are *Cold*, *Warm*, *Hot* and *HotPlusMirrored*.

a) Cold: In Cold NTR, only a single server is active at any given point in time.

b) Warm: In Warm NTR, redundant servers may be active but only a single server can connect to the downstream device(s). This is useful in situations where the device(s) can only support a single connection at a time.

c) Hot: In Hot NTR, all of the redundant servers are active and more than one server may be connected to the downstream device(s). The servers participating in a Hot NTR redundancy server set operate independently and are expected to have "minimal knowledge" on each other.

d) HotPlusMirrored: In HotPlusMirrored, also referred to as *Hot+* and *hot and mirrored*, all of the servers in a group mirror their internal states across each other. More than one server may be active and connected to a downstream device. The mirroring must at least include "Sessions, Subscriptions, registered Nodes, ContinuationPoints, sequence numbers, and sent Notifications".

IV. Coordination in OPC UA Server Redundancy

Based on the description given in Section III several aspects of server redundancy in OPC UA are in need of reliable coordination measures. This section presents these needs and details an integrated solution based on OPC UA and ZooKeeper to meet them.

A. Demands

To start by discerning the demands for coordination in OPC UA Server Redundancy, these include:

1) the requirement for a synchronised address space such that an identical hierarchy of nodes is exposed to connected clients. This includes identical Nodes, NodeIds,

browse paths, and address space structure. The only exempt nodes are ones in the local Server namespace that, for example, expose server diagnostics information. This need is universal across all failover modes. Further requirements exist for specific failover modes, such as the replication of unique identifiers for events across servers in TR or Hot+ configurations [6].
2) the second requirement is for a reliable mechanism for the detection of server failures. In the case of a TR server set, this would ensure the timely transfer of a session and its subscriptions to a functional server [6]. In NTR, this would allow for the automated start up of an application and/or connection to a downstream device. A service is therefore required to ensure failure detection and, if appropriate, contention resolution measures for the position of active server or to determine which server may connect to the downstream device.

The aforementioned requirements may be resolved by using ZooKeeper as a reliable configuration store for address space replication and as a central point for failure detection, leader election, and contention resolution. Using ZooKeeper for these services delivers several benefits to the system. The first is that any newly added OPC UA server only needs to be told how to connect to the ZooKeeper service and it may then download any other configuration information necessary, including its address space, and discern its role in the redundancy set [11]. The second benefit is derived from ZooKeeper's support for watches and notifications which allow OPC UA servers to subscribe to changes and undergo run-time reconfiguration [11]. Third, ZooKeeper's snapshots and logs may be used to recreate the state of the information model at any point in time for diagnostic purposes. Last of all, amongst ZooKeeper's obvious benefits are its guarantees for consistency, ordering, reliability, and its scalability born of its use of the Zab protocol and Observer servers, respectively.

In order to realise the above applications, certain requirements need to be imposed on the system:

- The entire address space of a redundancy server set must be stored on the distributed coordination service.
- Any modifications to the address space must be atomic and reflected on the coordination service.
- Any running server in a redundancy set must register its type and state on the distributed coordinator.
- For all of the above points, the distributed coordination service must be the only source of truth in the system.

To demonstrate this integration of OPC UA and ZooKeeper for distributed coordination, subsections IV-B to IV-D present the overall architecture of the resulting system, the data model used for ZooKeeper, the different architectural components, and their implementation details.

B. The zkUA Architecture

The ZooKeeper-OPC UA (zkUA) system architecture is composed of four software components shown in Fig. 1:

1) ZooKeeper Ensemble: Necessarily, the distributed coordination service in our scenario. Out of an ensemble of n servers, it is best practice for n to be an odd number and that a majority be used to form the quorum. Doing so would tolerate f servers crashing, where $f<n/2$, without it resulting in undesirable behaviour, e.g., split-brain scenarios [3].
2) zkUA Server: Every server participating in a redundancy set in the system must be integrated with ZooKeeper for several reasons. It ensures that any modifications to the address space locally or on ZooKeeper are reflected in the other. This allows redundant servers to provide connected OPC UA Clients with a homogeneous view of their address space. The integration is also needed for the correct operation of failure detection, leader election, and contention resolution for reasons that will be clear in section IV-D.
3) zkUA Proxy: For the migration of existing OPC UA servers to the zkUA system, a zkUA Proxy is required. This proxy is both an OPC UA and a ZooKeeper client. Its purpose is to replicate the address space present on an OPC UA Server to ZooKeeper. Replication of changes from an address space on ZooKeeper to legacy OPC UA Servers should not be supported as the legacy servers may include functions that would disrupt the overall behaviour of the zkUA system.
4) zkUA Failover Controller: This component, as the name implies, is the main management component in the architecture. It is responsible for detecting and reporting zkUA Server failures, initiating a failover, and participating in contention resolution and resource fencing. The details of this component is discussed in Subsection IV-D.

C. The ZooKeeper Data Model

A fifth component essential to the zkUA system is the data model deployed on ZooKeeper and shown in Table I. The first znode under the root node is the /Servers znode. This denotes the parent znode under which all OPC UA Server redundancy sets operate. Each redundancy set requires a neutral identifier under the /Servers znode where the set's address space may be stored and its activities organised. A globally unique identifier (GUID) is currently used to represent each redundancy set. In order to advertise the GUID via the OPC UA address space, a new Variable node, the *GroupGUID*, is created under the OPC UA-specified ServerRedundancy object. The GroupGUID's value is set to the GUID of the redundancy set. Both the path and the value are shown in Fig. 2.

Every Node is uniquely identified on ZooKeeper under the /AddressSpace path using its NodeId. The NodeId is converted into a string that holds the Node's namespace and identifier, e.g., a node belonging to namespace index 1 NodeId 3000 would be represented as "ns=1;i=3000" under the AddressSpace path. The znode's data is set with the encoded attributes of the Node it represents.

Naturally, this data model may be extended based on the failover mode to accommodate any additional information requiring synchronisation across the redundancy set. While

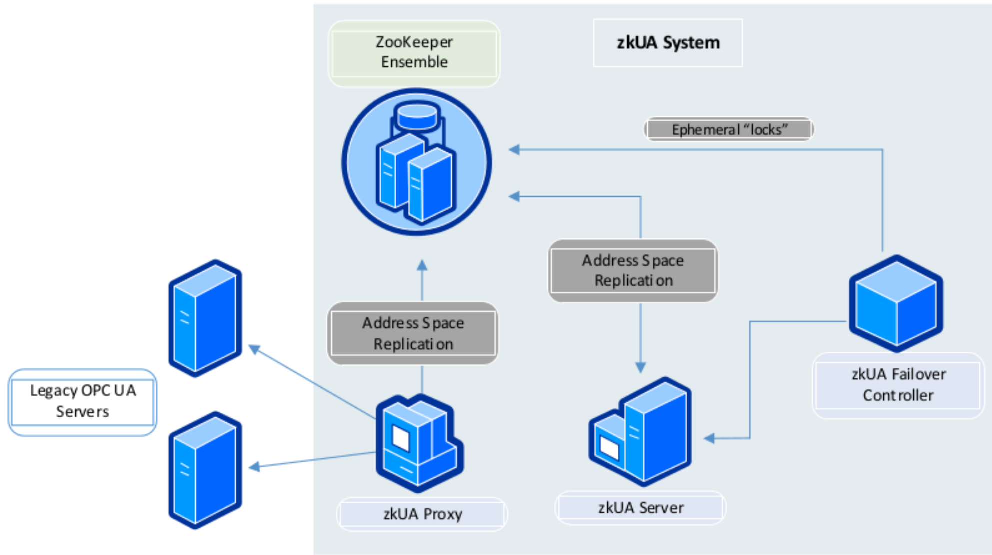

Fig. 1. The zkUA System Architecture

TABLE I
THE ZOOKEEPER DATA MODEL FOR OPC UA SERVERS REDUNDANCY

ZooKeeper Path	Type of zNode	Explanation
/Servers	Persistent	The root path for zkUA Server redundancy sets.
/Servers/{GroupGUID}	Persistent	Every redundancy set is assigned a unique GUID to differentiate it from the others.
/Servers/{GroupGUID}/AddressSpace	Persistent	This path stores all of the OPC UA Nodes with their respective attributes and references for storage, synchronisation, and replication.
/Servers/{GroupGUID}/Active	Persistent	If a zkUA Server is in a functional state, connected to a downstream device, and ready to serve clients, then it is in an active state and is represented by an ephemeral znode under this path.
/Servers/{GroupGUID}/{Failover Mode}	Persistent	All zkUA Servers that support a specific failover mode and are part of the same redundancy server set are each represented by an ephemeral znode under the correct Failover Mode path. Paths are available for Transparent, Cold, Warm, Hot, and Hot+ redundancy.

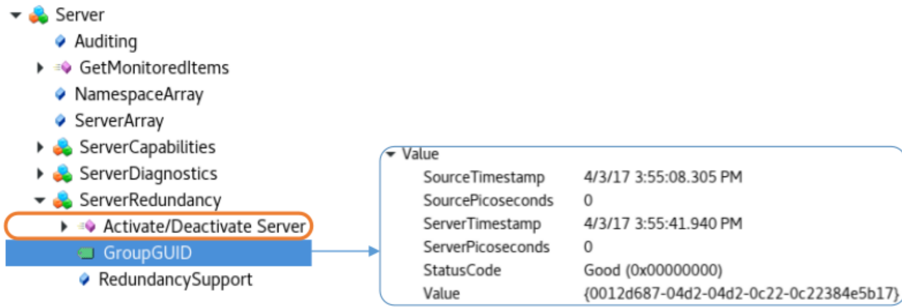

Fig. 2. The "GroupGUID" Variable Node and "Activate/Deactivate Server" Method Node

in its current state, however, it is apparent that an indirect result of having all zkUA Servers register on ZooKeeper is that ZooKeeper unwittingly doubles as a discovery service. The remainder of the data model will be clarified in the coming subsection as the operations of the zkUA components are discussed.

D. The zkUA Components

a) zkUA Proxy: To start with the zkUA Proxy, as previously mentioned, this component is developed as a migration path for non-ZooKeeper integrated OPC UA Servers. The component is composed of a ZooKeeper client and an OPC UA Client. The proxy is initialized with a configuration file that specifies several parameters listed in Table II. The proxy must be told which OPC UA Server's address space to replicate to ZooKeeper. It must also be told how to connect to it and the redundancy server set's GUID so that it may push the encoded address space to the correct path on ZooKeeper.

b) zkUA Server: The zkUA Server, similar to the zkUA Proxy, is initialized using a configuration file providing it with start up parameters such as its URI, port, redundancy group GUID, failover mode and role, as shown in Table II.

TABLE II
THE ZKUA START UP CONFIGURATION FILE PARAMETERS

Parameter	Component	Explanation
Hostname	Proxy/Server	The hostname belonging to the OPC UA server.
PortNumber	Proxy/Server	The port number the OPC UA Server is bound to.
GroupGUID	Proxy/Server	The GroupGUID representing the UA redundancy group on ZooKeeper.
Username	Proxy/Server	The Username used to login to the UA Server.
Password	Proxy/Server	The Password used to login to the UA Server.
ZkServer	Proxy/Server	The hostnames and port numbers of ZooKeeper servers participating in the ensemble.
RedundancyType	Server	The zkUA Server needs to be informed of its redundancyType. Options are: standalone, transparent, cold, warm, hot, and hot+.
State	Server	Instructs the zkUA Server if it is to start in an active or inactive state.
ServerId	Server	Required if the server is running in transparent redundancy mode, otherwise it is not.
AvailabilityPriority	Server	If set to true, the locally cached address space is used as a fallback for reads in case the server's connection to ZooKeeper is interrupted.

An additional parameter, *AvailabilityPriority*, is included in the configuration file to allow reads to continue from the zkUA server's local cache even if the communication between it and the ZooKeeper ensemble fails. Although this stops ZooKeeper from being the only source of truth, this may be required in cases where the continued uninterrupted availability of the zkUA Server is necessary. In such cases, it is advisable that modifications to the local cache be kept at a minimum, e.g., only permitting the continued polling and storage of values from downstream devices in the address space while disallowing any add/delete operations or the modification of Node types until the connection to ZooKeeper is restored.

A new Method Node, the "Activate/Deactivate Server", is also needed by every zkUA Server. Shown in Fig. 2, the Method and its associated internal function are used to modify the state of a zkUA Server when a failover is initiated.

c) zkUA Failover Controller: Finally, the zkUA Failover Controller is one of the most critical components in the system as it manages the behaviour and role of the zkUA Servers. The controller, similar to the proxy, is both an OPC UA Client and a ZooKeeper client. The controller is initialized with all of the parameters found in a zkUA Server configuration file except for the "AvailabilityPriority" parameter. These configurations are used by the controller to know which zkUA Server to monitor, how to connect to it, and what the correct failover mode and behaviour should be.

The controller is modelled on Hadoop's HDFS ZKFailoverController [12] and, therefore, performs:

- zkUA Server status monitoring: the controller of a specific zkUA Server periodically polls the server's state to determine its health. If the server responds in a timely manner with an acceptable state then it is considered to be healthy. Otherwise, it is not. The acceptable responses for the server's state differ with the failover mode.
- zkUA Server registration: Once initialized with a specific failover mode, the controller opens a session with ZooKeeper and creates an ephemeral znode under the redundancy group's path for that mode.
- zkUA contention resolution: If a zkUA Server is initialized in an active state and the failover mode supports more than one active server at a time, the controller creates an ephemeral znode under the redundancy set's Active path. If the failover mode or scenario only supports one active server at a time then only one controller and server combination is capable of creating an ephemeral znode under the Active path at a time, effectively acquiring a lock for the downstream device. The controller/server combo that is first to create the znode acquires the lock. All other controllers then monitor the lock for deletion. This typically occurs if the controller with the lock determines that its zkUA Server is in an unhealthy state. In such a case, the controller terminates its session with ZooKeeper. As the session expires, the ephemeral znodes are deleted, and all other controllers monitoring the znode are notified of the deletion event. The respective active controllers then try again to be the first to create a lock.

V. IMPLEMENTATION

A prototypical implementation[1] is made using the open source ZooKeeper[2] and open62541 [13] libraries. The resulting code achieves the requirements in Section IV by intercepting calls in the open62541 library to OPC UA Node addition, deletion, and modification (attribute writing) functions. Specifically, the functions are re-defined in the amalgamated open62541 library header file such that calls to the following functions are redirected to zkUA interception functions:

- Service_Write: This function is called once when an OPC UA Client modifies the attribute of a Node on an OPC UA Server over the network.
- UA_Server_Write: This function is called once when an OPC UA Server edits a Node's attribute.
- Service_AddNodes_single: This function is called once when an OPC UA Client or Server adds a new Node to the address space of the OPC UA Server.

[1]https://github.com/AGIsmail/zkUACoordination.git
[2]https://zookeeper.apache.org/

- Service_DeleteNodes_single: This function is called once when an OPC UA Client or Server deletes a Node from the OPC UA Server's address space.

For attribute changing operations, the intercepting function should save a copy of the current state of the Node to be modified before calling the original open62541 function. Once the original function finishes updating the local cache, the intercepting function then encodes the Node's NodeId, attributes, and its parent's NodeId and reference and pushes the encoded information to ZooKeeper. The parent's NodeId and reference are required to preserve the structure of the address space when replicating to another OPC UA Server. If the push fails, then the entire process should be reversed. The same process applies for the addition of a new Node, except for saving a copy of the original state of the Node as it should not yet exist in the local cache.

When a Node is modified and pushed to ZooKeeper, the remaining servers in the redundancy set receive a notification for the change in the znode's data. Since the Node may already exist on the zkUA Server, the node must be deleted and re-added. This is because the Node's type may have been modified in the process thereby invalidating the option of using attribute writing functions during replication. For these cases, Node deletion must not be replicated back to ZooKeeper and must only be enacted upon the local cache to prevent unwanted behaviour in the system.

Every Node that is added or modified in a zkUA Server is accompanied by its transaction zxid. The Node's znode path and zxid form key-value pairs that are stored in a hashtable local to each zkUA Server. The hashtable is then used to guarantee that the local zkUA Server is in sync with ZooKeeper and to prevent unnecessary operations to the address spaces stored on the zkUA Servers.

Read operations do not need to be redirected to ZooKeeper. By setting watches on the entire address space on ZooKeeper, the zkUA Server ensures that the local cache is always up to date. The only case in which this does not apply is when the zkUA Server's cache is being read while it suffers from an interrupted session. In such situations, reads from the local cache are forbidden unless the "AvailabilityPriority" parameter is set to true in the server configuration file for the special case where the availability of the zkUA Server is more important than the reliability of the served data.

As may be expected, further details related to the failover modes and other implementation specifics exist. However, the above suffices for the purposes of the discussion in the coming section. For further information, it is encouraged that the codebase be visited.

VI. Discussion

As may be apparent from the previous sections, the presented architecture and accompanying implementation ensure that the entire address space of a redundancy server set is stored on ZooKeeper. Any modifications to the address space are atomic in nature. All servers are registered on ZooKeeper and participate in failure detection, leader election, and contention resolution. The system is designed to treat ZooKeeper as the only source of truth, thereby avoiding split-brain scenarios.

One should note, however, the caveats involved in using such a system. First of all, while the scalability of reads are possible using ZooKeeper Observers, due to the per-watch-set memory penalty, the ZooKeeper service may require memory-abundant systems. Yet, since watches are stored locally on the ZooKeeper server that the zkUA Server connects to, this should not be a problem if the overall system is designed with enough ZooKeeper servers and hardware resources. The design should therefore reflect the expected overall number of server redundancy sets and their respective address space sizes.

A second point to mention in brief is that of security. While not addressed in this paper, ZooKeeper provides support for ACLs, as mentioned in Subsection II-H. In principle, this should prevent any unwanted manipulation of zkUA redundancy set behaviour. The system may be hardened further if communication between the ZooKeeper servers and the zkUA Servers, Proxies, and Failover Controllers are encrypted with SSL. Yet, since communication between the quorum is not encrypted, tunnelling may also be necessary.

Another point to mention in this section is related to an unintended, yet positive, consequence of the presented architecture. Since the ZooKeeper service is effectively storing the address spaces of all of the participating zkUA servers, ZooKeeper may be considered an active OPC UA Chaining Server. However, to exploit this situation a specially designed zkUA Client capable of reading directly from ZooKeeper would be required. Otherwise, a proxy that is both a ZooKeeper Client and a shell OPC UA Server may act as an interface to ZooKeeper for other OPC UA Clients.

One other detail to discuss is that of Method Node replication. While not present in our prototypical implementation, possible solutions to achieve this may require:

1) the functions associated with the Method Nodes to be available on all of the zkUA Servers in the entire system. This allows for a single generic zkUA Server to be used throughout the system. This may, however, result in a larger sized implementation which has a number of costs associated with it, e.g., a larger attack surface. Alternatively, each redundancy set may have its own flavour of zkUA Servers with the appropriate functions built. While potentially resulting in smaller implementations, this would increase the complexity involved in managing and developing for the system.
2) the use of function stubs. A possible implementation in this case may follow a service-oriented approach whereby a sub-system of services are created to represent the different methods to be called.

As may already be apparent from the architecture of IV-B, the zkUA Failover Controller, while critical to the system, is a single point of failure. A controller may crash while its zkUA Server remains functional. This vulnerability is mirrored in Hadoop's HDFS as well [12]. For mitigation, the controller should be monitored for unexpected failures and

restarted appropriately [12]. To ensure the orderly operation of zkUA Servers, a possible solution may include having active zkUA Servers watch for the disappearance of their ephemeral znodes on ZooKeeper and forfeiting their active roles when appropriate.

With respect to related work, while several other papers have addressed implementations for OPC UA Aggregation Servers [14, 15] or Historical Servers [16], there appears to be only one other publication addressing OPC UA Redundancy [10]. In [10], a different goal is pursued as the paper carries out a performance evaluation for OPC UA redundancy using the Java Client-Server SDK by Prosys. While this makes the presented system unique, it also deprives it of a basis for comparison.

VII. Conclusion

In this paper, the coordination requirements of OPC UA Server redundancy are quantified and shown to be realisable using the ZooKeeper service. An explanation of the overall architecture, data model, and components of the integrated OPC UA and ZooKeeper system is given. This includes the appropriate consideration of a solution for the migration of existing OPC UA systems. An example implementation based on the open source ZooKeeper and open62541 libraries is described in order to elaborate on implementation specifics.

While real-world deployments would still require careful design to ensure that sufficient resources are present for safe operation, the resulting system should be capable of providing a reliable framework for OPC UA Server redundancy. Through this system, redundant servers may achieve the required goals of synchronisation and replication, failure detection, failover initiation, and resource fencing. The extensibility of the data model given should present opportunities to accommodate further synchronisation requirements than those shown in this paper. It is expected that future iterations of this system address more complex features of OPC UA, such as Method Node replication, and to include more technologies from the IT domain to address other open questions in the standard, where identified and appropriate.

Acknowledgment

This paper is supported by TU Wien research funds as part of the Doctoral College Cyber-Physical Production Systems.

References

[1] Ahmed Ismail and Wolfgang Kastner. “Surveying the Features of Industrial SOAs”. In: *2017 Annual IEEE Industrial Electronics Society’s 18th International Conference on Industrial Technology (ICIT)*. Mar. 2017.

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

[3] Flavio Junqueira and Benjamin Reed. *ZooKeeper*. English. Sebastopol, CA: O’Reilly Media, 2013.

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

[5] Ailidani Ailijiang, Aleksey Charapko, et al. “Consensus in the Cloud: Paxos Systems Demystified”. In: *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. 2016.

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

[7] Hongchao Deng and Flavio Junqueira. *ZooKeeper SSL User Guide*. Atlassian Confluence. July 24, 2015. URL: https://cwiki.apache.org/confluence/display/ZOOKEEPER/ZooKeeper+SSL+User+Guide (visited on 04/06/2017).

[8] Ahmed Ismail and Wolfgang Kastner. “Vertical Integration in Industrial Enterprises and Distributed Middleware”. In: *International Journal of Internet Protocol Technology* 9.2/3 (2016), pp. 79–89.

[9] Ahmed Ismail and Wolfgang Kastner. “A middleware architecture for vertical integration”. In: *2016 1st International Workshop on Cyber-Physical Production Systems (CPPS)*. Apr. 2016.

[10] R. Cupek, K. Folkert, et al. “Performance evaluation of redundant OPC UA architecture for process control”. In: *Transactions of the Institute of Measurement and Control* (Sept. 2015).

[11] Sean Mackrory. *How-to: Use Apache ZooKeeper to Build Distributed Apps (and Why)*. Cloudera Inc. Feb. 14, 2013. URL: http://blog.cloudera.com/blog/2013/02/how-to-use-apache-zookeeper-to-build-distributed-apps-and-why/ (visited on 04/04/2017).

[12] *HDFS High Availability Using the Quorum Journal Manager*. URL: https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html (visited on 11/17/2016).

[13] F. Palm, S. Grüner, et al. “Open source as enabler for OPC UA in industrial automation”. In: *2015 IEEE ETFA*. Sept. 2015.

[14] Daniel Großmann, Markus Bregulla, et al. “OPC UA server aggregation—The foundation for an internet of portals”. In: *2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 2014.

[15] Ilkka Seilonen, Tomi Tuovinen, et al. “Aggregating OPC UA servers for monitoring manufacturing systems and mobile work machines”. In: *2016 IEEE ETFA*. 2016.

[16] Jukka Asikainen. “OPC UA Java History Gateway with Inherent Database Integration”. MA thesis. Aalto University, 2014.