## Redundancy

### Redundancy overview

Redundancy in OPC UA ensures that both Clients and Server can be redundant. OPC UA does not provide redundancy; it provides the data structures and services by which redundancy may be achieved in a standardized manner.

### Server redundancy overview

#### General

Server redundancy comes in two modes, transparent and non-transparent. By definition, in transparent redundancy the failover of *Server* responsibilities from one *Server* to another is transparent to the *Client*: the *Client* does not care or even know that failover has occurred; the *Client* does not need to do anything at all to keep data flowing. In contrast, non-transparent failover requires some activity on the part of the *Client*.

The *ServerRedundancy Object* defined in Part 5 indicates the mode supported by the *Server*. The *ServerRedundancyType* ObjectType and its subtypes *TransparentRedundancyType* and *NonTransparentRedundancyType* defined in Part 5 specify information for the supported redundancy mode.

The two areas where redundancy creates specific needs are in keeping the *Server* and *Client* information synchronised across *Servers*, and in controlling the failover of data flow from one *Server* to another.

Independent of the used redundancy mode it is expected that all *Servers* in the redundant set have an identical address space including identical *NodeIds* and the identical logic for setting the service level.

#### Transparent redundancy

For transparent redundancy, all that OPC UA provides is the data structures to allow the *Client* to identify which *Servers* are in the redundant set, what the service level of each *Server* is and which *Server* is currently responsible for the *Client* *Session*. This information is specified in *TransparentRedundancyType ObjectType* defined in Part 5.

All OPC UA interactions within a given session shall be supported by one *Server* and the *Client* is able to identify which *Server* that is, allowing a complete audit trail for the data. It is the responsibility of the *Servers* to ensure that information is synchronised between the *Servers*. A functional *Server* will take over the *Session* and *Subscriptions* from the failed *Server*. Failover may require a transport layer reconnect of the *Client* but the *Endpoint* URL of the *Server* shall not change. See 6.5 for more details on re-establishing connections.

Figure 25 shows a typical transparent redundancy setup.



Figure 1 – Transparent Redundancy setup

#### Non-transparent redundancy

##### Overview

For non-transparent redundancy, OPC UA provides the data structures to allow the *Client* to identify what *Servers* are available in the redundant set and also *Server* information which tells the *Client* what modes of failover the *Server* supports. This information allows the *Client* to determine what actions it may need to take in order to accomplish failover. This information is specified in *NonTransparentRedundancyType ObjectType* defined in Part 5.

Figure 26 shows a typical non-transparent redundancy setup.



Figure 2 – Non-Transparent Redundancy setup

For non-transparent redundancy the *Server* has additional concepts of *Cold*, *Warm*, *Hot* and *HotPlusMirrored* failover modes. The failover mode provides information about the failover capabilities the *Server* supports and allows a *Client* to determine the available failover actions described in Table 102.

All *Servers* in a redundant set shall have time synchronization.

##### *Cold*

From the *Servers* Perspective a cold system is where only 1 *Server* can be active at a time. This may mean that redundant *Servers* are unavailable (not powered up) or are available but not running (PC is running, but application is not started).

From the *Clients* Perspective a cold system is where the *Client* can only connect to 1 *Server* at a time. When the *Client* loses connectivity with the active *Server* it will attempt a connection to the redundant *Server*(s) which may or may not be available. In this situation the *Client* may need to wait for the redundant *Server* to become available and then

**Note**: may be

##### Warm

From the *Servers* perspective a *Warm* system is where the backup *Servers* can be active, but cannot connect to actual data points (typically, a system where the underlying devices are limited to a single connection). Underlying devices, such as PLCs, may have limited resources that permit a single Server connection. Therefore, only a single Server will be able to consume data. The *ServiceLevel* *Variable* defined in Part 5 indicates the ability of the *Server* to provide its data to the *Client*.

From the *Clients* perspective a Warm system is where the *Client* should connect to 1 or more Servers in the redundant set primarily to monitor the Service level.

**Note**: may be

##### Hot

From the *Servers* perspective a *Hot* system is where all *Servers* are powered-on and are up and running. In scenarios where *Servers* acquire data from a downstream device, such as a PLC, then one or more *Servers* are actively connected to the downstream device(s) in parallel. These *Servers* have minimal knowledge of the other *Servers* in their group and are independently functioning. When a *Server* fails or encounters a serious problem then its *ServiceLevel* drops. On recovery, the *Server* returns to the Redundant set with an appropriate *ServiceLevel* to indicate that it is a normal backup/redundant *Server*.

From the *Clients* perspective a *Hot* system is where the *Client* should connect to one or more *Servers* in the redundant set and to subscribe to the *ServiceLevel* variable (defined in Part 5) to find the highest service level to achieve load balancing; this means that *Clients* should issue commands (such as *Browse*, *Read*, *Write*, etc.) to the *Server* with the most availability. Subscription related activities will need to be invoked for each connected Server. Clients have the following choices for implementing subscription behaviour in a *Hot* system:

1. *Client* connects to multiple *Servers* and establishes subscription(s) in each where only one is *Reporting*; the others are *Sampling* only. On a fail-over the *Client* must enable *Reporting* on the next *Server* with the highest availability.
2. *Client* connects to multiple *Servers* and establishes subscription(s) in each where all subscriptions are *Reporting*. The *Client* is responsible for handling/processing multiple subscription streams concurrently.

*Clients* are not expected to automatically switch over to a *Server* that has recovered from a failure, but the *Client* should establish a connection to it.

##### HotPlusMirrored

From the *Servers* perspective a *HotPlusMirrored* system is where failovers are for *Servers* that are mirroring their internal states to all *Servers* in the redundant set and more than one *Server* can be active and fully operational. Mirroring state minimally includes *Sessions*, *Subscriptions*, registered *Nodes*, *ContinuationPoints*, sequence numbers, and sent *Notifications*. The *ServiceLevel* *Variable* defined in Part 5 should be used by the *Client* to find the *Servers* with the highest service level to achieve load balancing.

From the *Clients* perspective a *HotPlusMirrored* system is where a *Client* only connects to one *Server* in the redundancy set because the *Server* will share this session/state information with the other *Servers*. (to prevent excessive load on the Servers) mode On a fail-over the *Client* will simply create a new *SecureChannel* on an alternate *Server* and then call *ActivateSession*; all *Client* activities (browsing, subscriptions, history reads, etc.) will then resume.

(see 1.1.2.2 *Transparent redundancy*)

##### Client Behaviours

To enable clients to connect to all *Servers* in the list, each *Server* in the list shall provide the *ApplicationDescription* for all *Servers* in the redundant set through the *FindServers* *Service*. This information is needed by the *Client* to translate the *ServerUri* into information needed to connect to the other *Servers* in the redundant set. Therefore a *Client* needs to know only one of the redundant *Servers* to find the other *Servers* based on the provided information.

a*Client* actions for initial connections and s In the case of failover modes *Cold*, *Warm*, and *Hot*, a *Client* can always use a lesser failover mode than the *Server* supports. For example, the *Server* supports *Hot* failover mode and the *Client* can use the *Warm* actions. In the case of failover mode *HotPlusMirrored*, the *Client* shall not use a lesser mode as it would generate unnecessary load on the *Servers*.

Table 102 – Redundancy failover actions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Failover mode and *Client* options | Cold | Warm | Hot (a) | Hot (b) | HotPlusMirrored |
| On initial connection in addition to actions on active server: |  |  |  |  |  |
| Connect to more than one OPC UA *Server*. |  | X | X | X | Optional for status check |
| Creating *Subscriptions* and adding monitored items. |  | X | X | X |  |
| Activating sampling on the *Subscriptions.* |  |  | X | X |  |
| Activate publishing. |  |  |  | X |  |
| At Failover: |  |  |  |  |  |
| CreateSecureChannel to backup OPC UA *Server* | X |  |  |  | X |
| CreateSession on backup OPC UA *Server* | X |  |  |  |  |
| ActivateSession on backup OPC UA *Server* | X |  |  |  | X |
| Creating *Subscriptions* and adding monitored items. | X |  |  |  |  |
| Activating sampling on the *Subscriptions*. | X | X |  |  |  |
| Activate publishing. | X | X | X |  |  |

#### Transparent redundancy via proxy

A vendor can use the non-transparent redundancy features to create a *Server* proxy running on the *Client* machine to provide transparent redundancy to the client. This reduces the amount of functionality that shall be designed into the *Client* and to enable simpler *Clients* to take advantage of non-transparent redundancy. The *Server* proxy simply duplicates *Subscriptions* and modifications to *Subscriptions*, by passing the calls on to both *Servers*, but only enabling publishing and sampling on one *Server*. When the proxy detects a failure, it enables publishing and/or sampling on the backup *Server*, just as the *Client* would if it were a redundancy-aware *Client*.

Figure 27 shows the *Server* proxy used to provide transparent redundancy.



Figure 3 – Server proxy for transparent redundancy

### Client redundancy

*Client* redundancy is supported in OPC UA by the *TransferSubscriptions* call and by exposing *Client* information in the *Server* information structures. Since *Subscription* lifetime is not tied to the *Session* in which it was created, backup *Clients* can monitor the active *Client’s* *Session* with the *Server*, just as they would monitor any other data variable. If the active *Client* ceases to be active, the *Server* shall send a data update to any *Client* which has that variable monitored. Upon receiving such notification, a backup *Client* would then instruct the *Server* to transfer the *Subscriptions* to its own session. If the *Subscription* is crafted carefully, with sufficient resources to buffer data during the change-over, there need be no data loss from a *Client* failover.

OPC UA does not provide a standardized mechanism for conveying the *SessionId* and *SubscriptionIds* from the active *Client* to the backup *Clients*, but as long as the backup *Clients* know the *Client* name of the active *Client*, this information is readily available using the *SessionDiagnostics* and *SubscriptionDiagnostics* portions of the *ServerDiagnostics* data.

### Network redundancy

#### Overview

Redundant networks can be used with OPC UA in either transparent or non-transparent redundancy.

Network redundancy can be combined with *Server* and *Client* redundancy.

#### Transparent

In the transparent network use-case a single *Server* *Endpoint* can be reached through different network paths. This case is completely handled by the network infrastructure. The selected network path and failover are transparent to the *Client* and the *Server*.

Figure 4 Transparent Network Redundancy  


Examples:

* A physical appliance/device such as a router or gateway which automatically changes the network routing to maintain communications.
* A virtual adapter which automatically changes the network adapter to maintain communications.

#### Non-Transparent

In the non-transparent network use-case the *Server* provides different *Endpoints* for the different network paths. This requires both the *Server* and the *Client* to support multiple network connections. In this case the *Client* is responsible for selecting the *Endpoint* and for failover. For failover the normal reconnect scenario described in XXXXX can be used. Only the *SecureChannel* is created with another *Endpoint*. *Sessions* and *Subscriptions* can be reused.

Figure 5 Non-Transparent Network Redundancy  


The information about the different network paths is specified in *NonTransparentRedundancyType* *ObjectType* defined in Part 5.

### Redundancy Considerations

There are important considerations for a redundant system, particularly for synchronization:

* EventIds: Each UA *Server* in a *HotPlusMirrored* redundant set will need to synchronize EventIds to prevent a *Client* from mistakenly processing the same event multiple times simply because the *EventIds* are different. This is very important for Alarms & Conditions. For Cold, Warm, and Hot redundancy sets *Clients* must be able to handle *EventIds* that are not synchronized. Following any fail-over the *Client* must call *Refresh*.
* Timestamp (Source/Server): If a *Server* is exposing data from a downstream device (PLC, DCS etc.) then the *SourceTimestamp* and *ServerTimestamp* reported by all redundant *Servers* should match as closely as possible. *Clients* should favor the use of the *SourceTimestamp*.
* ContinuationPoints: Following a fail-over *Clients* must assume that any *ContinuationPoints* have been lost except in a *HotPlusMirrored* redundancy set.
* Methods: For *Clients* invoking *Methods* in *Servers,* following a failover *Clients* must determine the state of any method invocations to verify the method was processed completely.

### Manually Forcing Failover

A user may need to force a failover on the *Client*, perhaps to apply patches to a *Server* etc.

The *Server.RedundancyServer* should expose a method called “OverrideServiceLevel” that:

* Can have security applied to it to restrict user access

*Clients* must not connect to a *Server* when its Server.ServiceLevel property is set to 0.

Signature

**OverrideServiceLevel**(

[in] Bool serviceLevelOverride

);

|  |  |
| --- | --- |
| **Argument** | **Description** |
| serviceOverride | *Takes a BOOLEAN argument:*   * **True***:* Sets’ the Server.ServiceLevel property to 0. [within the range defined in 2.1] * **False**: This *Server* can be brought back into the redundancy rotation (perhaps as primary, or as a backup) and allows the *Server* to set Server.ServiceLevel property to an appropriate value. [within the range defined in 2.1 |

Method Result Codes (defined in Call Service)

|  |  |
| --- | --- |
| **Result Code** | **Description** |
| Bad\_UserAccessDenied | The current user is not authorized to invoke the method |
|  |  |

Table 96 specifies the *AddressSpace* representation for the *OverrideServiceLevel* *Method*.

Table 96 – OverrideServiceLevel Method AddressSpace Definition

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Attribute** | **Value** | | | | |
| BrowseName | OverrideServiceLevel | | | | |
| **References** | **NodeClass** | **BrowseName** | **DataType** | **TypeDefinition** | **ModellingRule** |
| HasProperty | Variable | InputArguments | Argument[] | PropertyType | Mandatory |

## Re-establishing connections

After a *Client* establishes a connection to a *Server* and creates a *Subscription*, the *Client* monitors the connection status. Figure 28 shows the steps to connect a *Client* to a *Server* and the general logic for reconnect handling. Not all possible error scenarios are covered.

The preferred mechanism for a *Client* to monitor the connection status is through the keep-alive of the *Subscription*. A *Client* should subscribe for the *State* *Variable* in the *ServerStatus* to detect shutdown or other failure states. If no *Subscription* is created or the *Server* does not support *Subscriptions*, the connection can be monitored by periodically reading the *State* *Variable*.



Figure 6 – Reconnect Sequence

When a *Client* loses the connection to the *Server*, the goal is to reconnect without losing information. To do this the *Client* shall re-establish the connection by creating a new *SecureChannel* and activating the *Session* with the *Service* *ActivateSession*. This assigns the new *SecureChannel* to the existing *Session* and allows the *Client* to reuse the *Session* and *Subscriptions* in the *Server*. This will result in the Client receiving data and event *Notifications* without losing information provided the queues in the MonitoredItems do not overflow.

The *Client* shall only create a new *Session* if *ActivateSession* fails. *TransferSubscriptions* is used to transfer the *Subscription* to the new *Session*. If *TransferSubscriptions* fails, the *Client* needs to create a new *Subscription*.

When the connection is lost, *Publish* responses may have been sent but not received by the *Client*.

After re-establishing the connection the *Client* shall call *Republish* in a loop, starting with the next expected sequence number and incrementing the sequence number until the *Server* returns the status Bad\_MessageNotAvailable. After receiving this status, the *Client* shall start sending *Publish* requests with the normal *Publish* handling. This sequence ensures that the lost *NotificationMessages* queued in the *Server* are not overwritten by new *Publish* responses.

If the *Client* detects missing sequence numbers in the *Publish* and is not able to get the lost *NotificationMessages* through *Republish*, the *Client* should read the values of all data *MonitoredItems* to make sure the *Client* has the latest values for all *MonitoredItems*.

Independent of the detailed recovery strategy, the *Client* should make sure that it does not overwrite newer data in the *Client* with older values provided through *Republish*.

If the *Republish* returns Bad\_SubscriptionIdInvalid, then the *Client* needs to create a new *Subscription*.

## Load Balancing

TODO

# Part 5 Modifications

## 6.3.1

**Original text**: “*ServiceLevel* describes the ability of the *Server* to provide its data to the client. The value range is from 0 to 255, where 0 indicates the worst and 255 indicates the best. The concrete values are vendor-specific. The intent is to provide the clients an indication of availability among redundant *Servers*.”

**Proposed text**: “*0-10 = do not use; 240+ = healthy/usable. More commentary needed; perhaps split by redundancy mode; provide use-cases etc.*”

## 6.3.9

**Original text**: “*RedundancySupport* is inherited from the *ServerRedundancyType*. It shall be set to COLD\_1, WARM\_1. HOT\_3 or HOT\_AND\_MIRRORED\_5 for all instances of the *NonTransparentRedundancyType*. It defines the redundancy support provided by the *Server*. The *Client* is allowed to access the redundant *Server* only as described there, however, ”hot” switchover implies the support of “warm” switchover and “warm” switchover implies the support of “cold” switchover. Support for HotAndMirrored redundancy implies the support of “hot” switchover, however, for Servers supporting HotandMirrored redundancy it is strongly recommended that *Clients* use the HotAndMirrored mechanisms.”

**Proposed text**: “”