Redundancy Reference

Configure redundant runtime environments.

Overview

The platform features redundancy for servers and data acquisition to deploy fault-tolerant systems and synchronization tools for solution consistency. It details configuring server redundancy within the software platform, setting up data redundancy, adjusting server parameters, and managing solutions in redundant setups. This page covers handling online and offline changes, hot starts, runtime modifications, and configuring clients based on redundancy, ensuring smooth redundancy management across clients.

Our platform has built-in features to deploy redundant systems, both at the server level and at the data acquisition level, allowing for the deployment of fault-tolerant systems in an easy and reliable process. Solution synchronization is a concern when using redundancy, but the platform has tools to simplify it by facilitating effective collaborative engineering.

Overview
Server Redundancy Types
High Availability Architecture
Redundancy Configuration
Replication Set
Solution Modification on Redundant System
Client configuration based on Redundancy scenario

Server Redundancy Types

There are two basic types of server redundancy approaches, “Hot-Hot” and “Hot-Standby”.

In most cases, the “Hot-Standby” is the preferred way to implement fault-tolerant systems.

Hot-Hot Redundancy

For a Hot-Hot scenario, two servers and all server modules are running at all times. A custom application (usually the TcpDataAccess driver) handles the synchronization of the operator inputs between the servers. When the application is feeding remote SQL databases or exchanging data with third-party systems, custom code in the application must ensure that data is not written twice.

The Hot-Hot configuration is used when the process does not allow any switching time from the primary to the secondary server. Therefore, there is no concept of an “active server.” Both servers are fully active at all times. The remote client's displays can access the solution from either server. To create Hot-Hot scenarios, you need to have two servers running, utilize the TcpDataAccess to synchronize any necessary data between the servers, and employ scripts to enable or disable parts of the running application as required by the duplicated operation.

The disadvantage of Hot-Hot implementations is that they require custom solution engineering, especially when integrating with other applications and when data acquisition from field devices is duplicated, which causes extra load on the network of devices and potentially results in differences in timestamps among each station.

How to manage two operators at each station trying to give commands at the same time, as well as the integration of remote SQL and file systems to avoid data duplication, must be decided at the application level.

For these reasons, the Hot-Standby configuration is the preferred option when implementing fault-tolerant servers. The Hot-Hot scenario is mainly applied when the server activation time of the hot-standby, usually from 1 second to a few seconds, is not acceptable for the process. Otherwise, Hot-Standby has a simpler configuration, requires no custom engineering, and automatically guarantees data consistency among the servers.

Hot-Standby Redundancy

In this scenario, there are two servers that have the solution loaded, but only one server (the ACTIVE one) is running all the modules. The STANDBY server has the modules in a PAUSED state and is receiving data synchronization from the ACTIVE computer to update its local memory but is not executing the tasks.

Redundancy is automatically implemented by the platform, using a simple configuration dialog. There is no need for custom programming or custom applications. Any solution created as a standalone solution can be deployed as a fault-tolerant pair with no engineering required, even when connected to third-party systems.

The communication between the servers uses the FrameworX ServiceClient protocol over TCP/IP, and can be encrypted. The data exchange uses a Tatsoft algorithm developed from years of field experience, which synchronizes the data using a publisher-subscriber model that sends data only by exception (events and data changes) in an efficient and reliable way.

Most importantly, the redundancy is not an external module; it is “redundancy-to-the-core.” Our platform was designed from the ground up to have built-in, kernel-level support for fault-tolerant applications.

When the STANDBY server detects that the ACTIVE computer is down, it changes the modules from the PAUSE state to the RUN state, starting with the real-time database's states and values according to how they have been received from the synchronization.

The detection of the Active Server is usually done by using a watchdog message on the same TCP/IP channel that is being used to exchange synchronization data, but custom switch determinations can be added if needed. The “Connection Timeout” parameter in the redundancy configuration defines the inactive time of the Active computer that triggers the standby computer to go live; the typical value is from one to ten seconds, depending on the network and servers.

High Availability Architecture

This section describes the system design, the configuration principles and the failover mechanism of a fault tolerant deployment. It is the architectural companion to the configuration procedures documented below.

Redundancy by layer

Redundancy is kernel level, not a bolt-on module. Each layer of the runtime participates:

Layer	How redundancy is achieved	On loss of a node
Communication and data acquisition	Both nodes hold the device configuration. The standby holds channels paused and receives values by synchronization	The promoted node opens its channels and resumes polling
Application and scripts	Modules are loaded on both nodes, running on the active and paused on the standby	Modules transition from PAUSE to RUN with the real time database already populated
Historian	Database redundancy replication keeps the standby database current, or both nodes use a centralized database. Store and Forward and database redundancy replication are mutually exclusive, for the Tag Historian and for alarm history, so choose one of the two per module	A standby node does not run Store and Forward. The Designer blocks Store and Forward and replication on the same module. If a solution has both, the runtime turns Store and Forward off at startup and logs the reason
Alarms	Alarm state is part of the synchronized set	Active alarms, acknowledgment state and shelving survive the switchover
Clients	Rich and Smart clients hold primary and secondary addresses	Rich and Smart clients redirect automatically and repaint from the promoted node. By default, HTML5 clients do not switch servers automatically: browser security constraints prevent a page from changing its server connection dynamically.

Failover sequence and timing

A switchover has three distinct phases. Figures published on different reference pages refer to different phases, which is why they differ.

Phase	What happens	Typical	Governed by
Detection	The standby stops receiving the watchdog message on the synchronization channel	1 to 10 s	`Connection Timeout`, configurable
Activation	Modules move from PAUSE to RUN, starting from the already synchronized real time database	under 1 s	volume of state, hardware
Client redirect	Clients reconnect to the promoted node and repaint	2 to 5 s	client count, display complexity, network

End to end, a typical deployment completes a switchover in under 5 seconds. The dominant term is Detection, and it is a tuning decision rather than a product limit: a shorter `Connection Timeout` shortens failover but raises the risk of a false promotion on a congested network. Tune it to the measured jitter of the link between the two nodes.

The activation figure assumes the standby has been receiving synchronization normally. A node that has just joined the pair completes its initial synchronization before it can be considered a valid failover target.

No single point of failure

No core service depends on a single node. Communication, application, historian and alarm processing are all present on both nodes, and the synchronization channel itself is the only shared element. That channel carries a watchdog, so its loss is treated as a failover condition rather than a silent degradation. Where the network is itself a risk, the synchronization link should not share physical media with the device network.

Geographically separated deployment

The synchronization channel runs over TCP/IP and can run over any routable link, including one between separate data centers. Two constraints apply. `Connection Timeout` must be tuned to the latency and jitter of the inter site link rather than left at a LAN default, otherwise normal WAN variation will trigger spurious failovers. And the historian strategy must be chosen deliberately: a centralized database at one site reintroduces a single point of failure that server redundancy alone does not remove.

Redundancy Configuration

The Hot-Standby redundancy is configured in the **Runtime / Startup / Redundancy Enabled** dialog. (You may need to enable **Advanced settings** checkbox)

Server Redundancy Configuration

The key parameters to configure are:

Timeout: Watch-dog time in seconds. This is the time the standby computer uses to monitor the activity of the active computer while doing the automatic switch. The Switch between active and standby can also be by command. In this case, the switch is immediate. Only the time to change the modules from Pause to Run on the standby computer is needed. The time ranges typically from one second to five seconds but is very dependent on specific Solution configuration, the server computers, and the network.

Primary and Secondary Server IP/Computer names: The computer IP address or DNS names of the computers that will work together as a fault-tolerant pair. They will run the exact same Solution configuration, and either one can be ACTIVE or STANDBY at any given time. In practice one of the two always starts first and takes the ACTIVE role; when the second node comes up, the pair resolves the roles according to what is configured in the Designer. That setting is the only difference between the computer designated as the Primary and the one designated as the Secondary:

- **On Primary Startup**: When the Secondary is the ACTIVE server and the Primary computer is down, the Primary computer starts. You can specify if the Secondary will stay as the ACTIVE server, or if the Primary server will take over the ACTIVE state after it finishes its initialization, which will automatically change the Secondary computer to the STANDBY role. The configuration is using the “On Primary Startup” option.

Based on these settings, a custom command line for the TStartup.exe is created and presented on the configuration line. When you are deploying redundant configuration, you need to use the command line on Windows AutoStart or on the Windows Service configuration if you are running the platform as a Windows Service.

You can configure application redundancy by configuring two computers to be used as servers. One computer will be the primary server and the other will be the secondary (hot-standby or hot-hot redundancy). If the primary computer or the connection to the computer fails, the system automatically fails over to the secondary computer.

To configure redundancy:

- Go to **Runtime / Startup / Redundancy Enabled**.
- Enter or select the information, as needed.

Field	Description
Redundancy Enabled	Select to enable the redundancy configuration.
Primary Server IP and Port	Enter the IP address and port of the primary server.
Secondary Server IP and Port	Enter the IP address of the secondary server. Both servers of a redundant pair use the same port. From FrameworX 10.1.6, Designer keeps the secondary port read-only and equal to the primary port. In earlier versions Designer accepted two different ports, and a pair configured that way had no working failover and showed no error. From 10.1.6, Designer reports a solution saved with two different ports, or with no secondary port, in the Output window when the solution opens, and the runtime launcher writes a warning to its trace. On the Production and Validation profiles, turn **Auto Ports** off for a redundant pair: when its port is busy, a server can move to another port, and its partner then cannot reach it. Designer flags Auto Ports on those profiles while redundancy is enabled. Development and Custom allow Auto Ports with redundancy.
On Primary Startup	Select the option you want.
Replication	Select how to handle historian replication. AlarmHistorian, TagHistorian and Retentive.
Timeout	Connection timeout time, in seconds. If reached, this will cause the system to switch to the secondary server.
Startup Command	Read-only field populated based on the fields above. Click **Copy to Clipboard** to copy the command for use.
RichClient CmdLine	Read-only field populated based on the fields above. Click **Copy** to copy the command for use.

Smart Client URL	Read-only field populated based on the fields above. Click **Copy** to copy the command for use.
HTML5 Client URL	Read-only field populated based on the fields above. Click **Copy** to copy the command for use.

Role Resolution at Startup

With redundancy enabled, a node has to know which of the two roles it holds before the server can start. The startup process compares the local addresses of the machine it is running on with the configured Primary and Secondary addresses. The address that matches determines the role: a machine whose own address is IP1 starts as the Primary, a machine whose own address is IP2 starts as the Secondary.

This is a local check, performed against the TCP/IP stack of the machine itself. It does not contact the peer node and it does not contact any network equipment. Only after the role is resolved are the redundancy parameters handed to the server, and only then does the negotiation with the peer begin.

If neither configured address is local, the node cannot hold a role in the pair. The redundancy parameters are withheld and the runtime starts as a single standalone node. This is deliberate: it prevents a machine that is neither IP1 nor IP2 from taking the place of one of the two nodes.

The resolved identity is fixed at startup. It determines the address and port the server listens on, which node is active, which node writes to the Historian, and how state and retentive tags synchronize with the peer. It is not migrated while the runtime is running, so a node that started standalone has to be restarted once the condition that prevented role resolution is corrected.

> ⚠ **When the pair does not form**
>
> A node that starts standalone because no configured address was local runs normally and reports itself as the active node. From *Framewor X 10.1.5e* this condition is recorded in the Windows Application Event Log and in the trace log files. It is not raised in the interface, so the Event Log is where to confirm it. In earlier versions the condition was discarded silently.

When a configured address is not local

Two situations account for nearly every occurrence:

- **The network is not ready yet.** On a service start right after a computer reboot, the network adapter may not have its address resolved and bound when the service starts. Link, negotiation and, with DHCP, the full lease cycle have to complete first. See *Starting as a Windows Service* below.
- **The interface has no link.** When an interface loses physical link, Windows removes the IP address of that interface from the stack, the behavior known as media sensing. The configured address then stops being local and role resolution fails by the same path. The runtime does not need to talk to the switch; the interface needs link for Windows to keep the address bound to it.

Inside a Docker container, local means local to the container and not to the Docker host. See Redundancy in Docker Bridge Networks.

Starting as a Windows Service

A redundant node is normally started by the Windows service created from **Runtime / Startup**. Because role resolution reads the local addresses of the machine, the service must not start before the network adapter has its address bound.

Register the service as Automatic (Delayed Start), with the Windows default delay. This is the installer default and the recommended configuration for a redundant node. The delay gives the machine time to complete link, negotiation and address assignment before the runtime starts.

Three adjustments are sometimes proposed to make the runtime start sooner. None of them is supported for a redundant node:

- **Plain Automatic start.** The service starts before the network is usable, role resolution fails, and the node starts standalone.
- **Reducing the delayed-start timer.** The Windows default of 120 seconds applies to the whole machine. Shortening it removes the margin this configuration exists to provide and moves the behavior toward a plain Automatic start.
- **A service dependency on the network.** Tcpip and Afd are boot-start drivers, loaded before the Service Control Manager processes Automatic services, so such a dependency is already satisfied and gains nothing. NSI and Dhcp are real services, but reaching the RUNNING state means the component started, not that an address is configured and bound to an adapter. The Service Control Manager has no concept of the network being usable.

On a machine where the network genuinely takes longer to come up, use `/redundancywaittimeout` (see *Server Parameters* below) rather than changing the start type.

> ⓘ From *FrameworX 10.1.5e*, the resolution of the local addresses of the machine is corrected for the first 60 seconds after a Windows boot, a window in which the address list could previously be returned unresolved even with the network already operational. The correction removes that failure, but it cannot create an address that the adapter does not yet have: a runtime that starts before the network is ready still cannot resolve its role.

Data Redundancy Configuration

When running fault-tolerant applications, the Alarm Archiving server and the Historian Archiving server can be located on the same computer as the platform's servers or on a third computer dedicated to archiving, such as a Microsoft SQL Server or Oracle server. The “Historian Replication” configuration provides support for many scenarios.

When the Historian is on a remote database machine, there is no need to enable replication on the platform. Since only one computer is ACTIVE, either the Primary or Secondary will write to the external database. In this case, you define “no replication” for the configuration.

Note that the external database can still be a fault-tolerant cluster that uses the database redundancy tools, but the cluster is viewed by the Solution as just one external connection.

If you are running either the Alarm database or the HistorianTags database on the same computer as the platform's servers, you need to enable the respective replication option.

Replication Set

When a Hot-Standby pair runs, the ACTIVE server keeps the STANDBY server consistent through two independent mechanisms: a real-time object-update channel for live values, and a set of background database-synchronization threads for stored data. It is useful to understand exactly what each one covers.

Live values vs. stored data

Live tag values, qualities and timestamps are **not** carried by the database-replication options below. They are propagated continuously over the publisher-subscriber object-update channel (the same by-exception synchronization described under *Hot-Standby Redundancy* above), so that when the STANDBY takes over, its real-time database already holds current values. The replication options described here govern only the persisted **databases**, not the in-memory tag values.

What the database synchronization covers

The ACTIVE server runs background synchronization for up to four stores. Two of them are opt-in, and two are always synchronized whenever redundancy is running:

Store	Synchronized	Notes
Alarm Historian	**Opt-in** — off by default	Replicates the alarm/event history rows. Enable with the **Alarm Historian** replication option (the `RedundancyReplicationAlarms` startup setting, default *false*). From FrameworX 10.1.6, rows also go from the STANDBY server to the ACTIVE server. See *Recovering history from the STANDBY server* below.
Tag Historian	**Opt-in** — off by default	Replicates the tag **historian** — the stored history rows of trended tags — **not** the live tag values. Enable with the **Tag Historian** replication option (the `RedundancyReplicationHistorian` startup setting, default *false*). From FrameworX 10.1.6, rows also go from the STANDBY server to the ACTIVE server. See *Recovering history from the STANDBY server* below.
Retentive	**Always**	The retentive-value database is synchronized whenever redundancy is running, so retentive tag values survive a failover. No option is required. Replication runs from the ACTIVE server to the STANDBY server only.
Runtime Users	**Always**	Runtime users created or edited at run time (the RuntimeUsers store) are synchronized to the pair whenever redundancy is running, so logins stay consistent across a switch. No option is required. Replication runs from the ACTIVE server to the STANDBY server only.

The two opt-in options are only relevant when the corresponding database (Alarm or Historian) is hosted **on the platform server itself**. When the Historian or Alarm database lives on a separate/remote database machine, replication is not needed — only the single ACTIVE server writes to it — so you leave those options off, as described under *Data Redundancy Configuration* above.

Tuning replication throughput on large projects

Available from FrameworX 10.1.5d.

On large redundant projects where the Alarm Historian or Tag Historian is replicated (the two opt-in options above) and the database is hosted **on the platform server itself**, the background synchronization must keep pace with the write rate so the STANDBY database stays current. Two keys in `TServer.json` (in the `C:\Users\Public\Documents\FrameworX\MachineSettings` folder), under the `appSettings` object's `Redundancy` section, tune this replication drain. They are independent of the Store and Forward batch settings (see *Historian Engine Reference Configuring Store and Forward Batch Sizes*), which govern a different synchronization path.

Key	Shipped value	Drives
`Redundancy.NotSyncBatchSize`	2000	Number of unsynchronized rows the ACTIVE server pulls per replication-sync pass, for both the Alarm Historian and Tag Historian replication drains. Increase it on high-volume projects (for example a historian with tens of thousands of trended tags) so the STANDBY backlog drains at the write rate instead of falling behind.

`Redundancy.UseFullRowNotSyncUpdate`	false	Controls how replicated rows are flagged as synchronized on the ACTIVE database. By default the engine issues a narrow, key-based update that touches only the synchronization flag — this avoids a full-row rewrite, which on compressed time-series stores such as TimescaleDB would otherwise force chunk decompression and dominate the sync cost. Set to `true` only to revert to the legacy full-row update for diagnostic or rollback purposes.

The shipped `TServer.json` sets `NotSyncBatchSize` to 2000. When the key is missing, or its value is outside the range [50, 50000], the engine uses 250. Both keys are optional.

How to change these settings

1. Stop all running solutions on the machine (each solution runs its own TServer process, but all of them read from the same `TServer.json` configuration file).
2. Open `TServer.json` in the `C:\Users\Public\Documents\FrameworX\MachineSettings` folder.
3. Inside the `appSettings` object, add or locate a `Redundancy` section and set the values:

```
"appSettings": {
  "Redundancy": {
    "NotSyncBatchSize": 2000,
    "UseFullRowNotSyncUpdate": false,
    "DisableReverseReplication": false,
    "ReverseNotSyncBatchSize": 0,
    "ReverseIntervalSeconds": 10,
    "ReverseInterSliceDelayMs": 250
  }
}
```

4. Save the file and restart the solutions.

The example also shows the four recovery keys described under *Recovery settings* below. The runtime reads `TServer.json` from the `MachineSettings` folder. The copies in the installation folder apply only when the `MachineSettings` file does not exist. Installing a new version or a patch keeps the existing `MachineSettings` file, which all installed FrameworX versions share, so keys added in a later version are missing from it until you add them.

Automatic failback and replicated history

Only the ACTIVE node collects data. When Alarm Historian or Tag Historian replication is enabled, the rows it records are replicated to the STANDBY node. While the Primary is down, the Secondary is ACTIVE and keeps recording; those rows cannot be replicated until the Primary is back.

With **On Primary Startup** set to return the ACTIVE role to the Primary (the `RedundancyOnStartupSwitchToPrimary` setting, command-line `/autoswitch`), the restarting Primary runs as STANDBY for the TimeAutoSwitch time and receives these rows from the ACTIVE Secondary during that time. When the time expires, the Primary becomes ACTIVE and the Secondary becomes STANDBY.

Rows the Secondary has not sent by then remain in the Secondary's database. From FrameworX 10.1.6, the Secondary sends them to the ACTIVE Primary while it is STANDBY, for the history selected for replication. See *Recovering history from the STANDBY server* below. In 10.1.5, a STANDBY node does not send its own rows. They are sent the next time the Secondary is ACTIVE. To send them in 10.1.5:

1. Run `@Server.Redundancy.SwitchToStandby()` on the ACTIVE Primary. The Secondary becomes ACTIVE and sends the remaining rows.
2. When they have been sent, run `@Server.Redundancy.SwitchToStandby()` again to return the ACTIVE role to the Primary.

/autoswitch is evaluated only when the Solution starts. A manual switch only swaps the ACTIVE and STANDBY roles.

Recovering history from the STANDBY server

Available from FrameworX 10.1.6.

Before 10.1.6, history replication ran in one direction only, from the ACTIVE server to the STANDBY server. Rows a server recorded while ACTIVE, and had not replicated when the roles switched, stayed on the recording server until it was ACTIVE again. Until then, trends, reports and queries answered by the ACTIVE server missed those rows.

From 10.1.6, the server in the STANDBY role sends those rows to the ACTIVE server, which stores them in its own Alarm Historian or Tag Historian database. This applies to whichever server is STANDBY, Primary or Secondary. The ACTIVE server compares each received row with the rows it already holds for the same time and skips a row whose content, apart from its ID, is identical. It stores each new row with an ID of its own and marks the row as synchronized, so the row never travels back.

A STANDBY server receiving Tag Historian replication records no tag history samples of its own. Its Tag Historian holds the history replicated from the ACTIVE server.

Requirements

- Both servers run FrameworX 10.1.6 or later.
- The **Alarm Historian** or **Tag Historian** replication option is on for both servers. Recovery covers only the streams selected for replication.

- `Redundancy.DisableReverseReplication` is `false` on both servers. This is the shipped value, so recovery needs no change to this key.
- Each server reaches its history database through a direct database connection to SQL Server, PostgreSQL, MySQL or SQLite. The first column of the history table is an auto-increment `ID` column: `IDENTITY` on SQL Server, `BIGSERIAL` on PostgreSQL, `AUTO_INCREMENT` on MySQL, `INTEGER PRIMARY KEY` on SQLite, with or without `AUTOINCREMENT`. FrameworX creates this column when it creates the table through a SqlClient, Npgsql, MySQL or SQLite connection. A table it creates through an ODBC or OleDb connection to SQL Server or PostgreSQL has a plain ID column and does not meet this requirement. This applies to the tables on both servers, because each server receives rows while it is ACTIVE.

When a table does not meet the `ID` requirement, the ACTIVE server recovers no rows into it and writes a warning, at most once a day, to `SyncHistorian.txt` in the `C:\Users\Public\Documents\FrameworX\TraceLogs` folder. The warning contains `Tags Historian reverse replication refused` or `Alarm reverse replication refused` and names the table and the reason. Normal replication from the ACTIVE server to the STANDBY server continues. With debug logging off, recovery writes nothing to the log when it stores every row it receives.

Recovery speed

Recovery sends smaller pieces than normal replication, with pauses between them, to limit the extra load on the ACTIVE server, which stores the recovered rows while it serves clients. Each pass reads, newest first, up to `ReverseNotSyncBatchSize` rows from each history table, or half of `NotSyncBatchSize` when `ReverseNotSyncBatchSize` is 0. It sends them in pieces of up to 200 rows, with a pause of `ReverseInterSliceDelayMs` between pieces. The next pass starts `ReverseIntervalSeconds` after the previous pass ends. While passes deliver nothing, the wait between passes grows step by step up to six times `ReverseIntervalSeconds`, with a maximum of five minutes. The back-off never shortens a `ReverseIntervalSeconds` above five minutes.

With the shipped values, each pass sends up to 1,000 rows from each table, and a table with a backlog recovers about 1,000 rows every 12 seconds. Tag Historian tables recover one after another within a pass, so the rate for each table falls as the number of tables grows. Alarm Historian and Tag Historian recover in parallel. To recover a large backlog sooner, raise `ReverseNotSyncBatchSize`. Larger passes add read work on the STANDBY server and insert work on the ACTIVE server while the backlog drains.

First recovery on Normalized tables

A Normalized Tag Historian table marks every new sample as pending replication, also while redundancy or Tag Historian replication is off. A server running without a redundant pair, or with Tag Historian replication off, therefore keeps its whole Normalized history pending. When it enters the STANDBY role with Tag Historian replication on, its first recovery sends all of it to the ACTIVE server, newest first. At the speed above, one day of 100 tags sampled every second, 8.64 million rows, takes about 30 hours. If the same server is ACTIVE first, normal replication sends those rows to the STANDBY server instead.

Recovery settings

Four keys in the same `appSettings` `Redundancy` section of `TServer.json` control recovery. Change them on both servers, as described in *How to change these settings* above. A server whose `MachineSettings` file comes from an earlier version has none of these keys and runs with the defaults below. When the file also has no `NotSyncBatchSize`, the engine uses 250, and a recovery pass reads 125 rows.

Key	Default	Range	Effect
`Redundancy.DisableReverseReplication`	`false`	`true`, `false`	`true` turns recovery off. A server set to `true` does not send while STANDBY and does not accept while ACTIVE, so setting it on one server stops recovery for the pair. Set the same value on both servers.
`Redundancy.ReverseNotSyncBatchSize`	0	0, or 10 to 25000	Rows read from each table in one pass. 0 uses half of `NotSyncBatchSize`, with a minimum of 10. With the shipped `NotSyncBatchSize` of 2000, a pass reads 1,000 rows.
`Redundancy.ReverseIntervalSeconds`	10	5 to 3600	Seconds from the end of one pass to the start of the next.
`Redundancy.ReverseInterSliceDelayMs`	250	0 to 10000	Pause in milliseconds between the 200-row pieces of a pass. 0 removes the pause.

The runtime ignores a value outside its range and uses the default. It reads these keys at startup only.

Pending data flush on switchover

Only the ACTIVE server writes alarm history and tag history. The STANDBY server keeps its Alarm and Historian modules paused until it takes over. When the ACTIVE server leaves the ACTIVE role, at a switchover or at a shutdown with redundancy enabled, it first runs a flush: it writes the data still pending in memory before it steps down. Two runtime properties let you skip this flush. Both default to `false`, which runs the flush.

Property	false (default)	true
`@Server.Redundancy.HistorianDisableFlush`	The Historian processes the tag changes still queued, for up to 2 minutes, then writes all accumulated samples to the historian tables.	The Historian does not process the queued tag changes. The samples already accumulated in the historian tables are still written. Only the tag changes not yet turned into samples are not recorded by this server.

`@Server.Redundancy.AlarmDisableFlush`	The Alarm module processes the queued events, for up to 2 minutes, then saves the pending active-alarm list and writes the pending alarm history and audit trail records to the database.	The Alarm module skips these steps, so the pending alarm history records are not written at the switchover. A database write already in progress is still completed, for up to 10 minutes. At a shutdown, this property applies only when redundancy is enabled.

The server that takes over does not receive the pending data of the server that stepped down. It writes from its own values, and it rebuilds its active-alarm list from its current state, so the active alarms are consistent after the switch. Alarm history records left pending by a disabled flush stay in the memory of the server that stepped down, and that server writes them, with their original timestamps, the next time it becomes ACTIVE.

Keep both properties `false` unless a faster switchover matters more than complete history. With `true`, data pending on the server that steps down can be missing from the history, or written later.

Application hooks on activation and deactivation

To run solution-specific logic when a node takes over or steps down during a switch, implement these optional methods on the server-side **ServerMain** script class:

- `OnActivate(bool)`: invoked on the node that is becoming ACTIVE, **just before** it takes over the ACTIVE role.
- `OnDeactivate()`: invoked on the node that is stepping down, as it leaves the ACTIVE role and returns to STANDBY.

Both are optional. If the method is not present on **ServerMain**, nothing is called.

What to do in OnActivate

Use `OnActivate` to set tag values, for example the initial values the solution needs when this node takes over. This is the supported use.

```
public void OnActivate(bool flag)
{
    @Tag.Line1.Mode.Value = 1;
    @Tag.Line1.StartedByFailover.Value = flag;
}
```

> `OnActivate` runs before the node is ACTIVE, and the Script module of that node can still be paused at that moment. A call that depends on the Script module, such as `TK.ExecuteClassMethodOnServerAsync` to run a method of another server class, may therefore not execute. Keep the method short and limit it to setting tag values.

The OnActivate parameter

The boolean parameter of `OnActivate` tells the incoming node *why* the switch is happening:

Value	Meaning
`true`	The node that was ACTIVE was shut down or could not be reached. The STANDBY is taking over as an automatic failover.
`false`	The switch was triggered by an explicit command (for example, `@Server.Redundancy.SwitchToStandby()`), not by a failure of the ACTIVE node.

Use it to set different tag values on an unplanned failover (`true`) than on a commanded switch (`false`), as the `StartedByFailover` tag in the example above does.

Server Parameters

When using redundancy, there is a set of parameters that can be assigned to both TStartup.exe and TServer.exe to specify its behavior. The command line is automatically created on **Runtime / Startup / Command Lines and URL / Startup Command**, but you can customize the command lines directly as needed. The parameters below are the redundancy-related ones. For the complete list of runtime startup parameters, including the logging, splash and diagnostic options, see Runtime Installation Reference.

- /ip1: <Primary Server Name or IP>
- /ip2: <Secondary Server Name or IP>
- /Port1: <Port number of primary, default is 3101>
- /Port2: <port number for secondary, default is 3101; must be the same as /Port1>
- /Solution: <full path of the Solution file>
- /connectiontimeout: <watch-dog timeout in seconds, accepts decimal points, default is 3>. This is the command-line form of the **Timeout** field of the redundancy dialog. It is a server-side parameter: it sets how long the standby node waits without the watchdog of the active node before taking over.
- /username: <startup user>
- /redundancy (has no parameters, just need to be included to enable redundancy)
- /redundancywaittimeout: <seconds to wait for the network stack on a Windows Service start, default is 0>. When the runtime starts as a Windows Service with /redundancy and neither the primary nor the secondary address is local yet — typical on a service auto-start right after a computer reboot — this option makes the startup wait up to the given number of seconds, re-checking every 5 seconds, and continue with

redundancy enabled as soon as one of the configured addresses becomes local. With the default of 0 the startup does not wait: the condition is recorded in the Windows Application Event Log and the runtime starts with redundancy disabled, as in previous versions. *Available from FrameworX 10.1.5e.*

- /autoswitch (has no parameters. if included, the Primary takes over as the Active node if the secondary was acting as Active)
- /TimeAutoSwitch: <number of seconds the Primary waits before becoming active if the autoswitch option is enabled. Usually set to 60 seconds.
- /SolutionIPPathName: <IP>;<Path of the Solution on the remote server>. The SolutionIPPathName is used by the system to allow one station to automatically update the Solution in the redundant pair when doing online Solution changes and HotStart commands. Example: /SolutionIP PathName:192.168.0.1;C:\FrameworX\Solutions\Example.dbsln
- /RedundantPairWebServicesPort: <TWebServices port number of the other node in the redundant pair, default is 10108>. Used together with /SolutionIPPathName: when one station updates the Solution on the redundant pair during online Solution changes and HotStart commands, it reaches the remote station's TWebServices on this port. Set it only when TWebServices on the remote server is configured to a port other than the default. Example: /RedundantPairWebServicesPort:80

The TimeAutoSwitch time applies only when you use the **/autoswitch** option. /autoswitch is evaluated when the Solution starts: if the computer designated as the Primary starts while the Secondary is ACTIVE, the Primary starts as STANDBY and takes over the ACTIVE role after the TimeAutoSwitch time (default 60 seconds).

While it waits, the Primary is the STANDBY node, so it receives from the ACTIVE Secondary everything a STANDBY receives, including the Alarm Historian and Tag Historian rows the Secondary recorded while the Primary was down, when those replication options are enabled. The switch happens when the time expires; it does not wait for that replication to finish.

When Alarm Historian or Tag Historian replication is enabled, increase the value beyond the default for large Solutions, slow networks, or high alarm and data-logging rates. There is no fixed rule for the value: it depends on the Solution architecture, the network and the data volume, so test it on your installation.

TimeAutoSwitch has no field in the redundancy configuration dialog. Add `/TimeAutoSwitch:<seconds>` to the TStartup command line you deploy (Windows AutoStart or the Windows Service configuration). See *Automatic failback and replicated history* above.

Full RichClient example:"<productPath>\fx-10\TRichClient.exe" /ip1:192.168.1.1 /ip2:192.168.1.2 /Port1:3101 /Port2:3101
Full SmartClient example:http://192.168.1.1:3101/smartclient?ip1=192.168.1.1&ip2=192.168.1.2&Port1=3101&Port2=3101

Solution Modification on Redundant System

The centralized Solution configuration makes it easy to keep a Solution synchronized on both servers. All the Solution settings are only in one of two files: the .dbsln and .dbrun file. You just need to make sure the file is the same on both computers when deploying your Solution.

Note that the switch depends on many factors: if there are pending operations, the Solution size, the computers, and the network. The total switch time is typically measured in seconds, but it is necessary to conduct a test on your specific scenario to specify the right parameters for the connection timeout and retry parameters.

You can set up your own procedure to synchronize the two files, or use the platform's automated methods for hot-standby configuration.

Online changes

Option 1: Local IP acting as the Primary Server

- When you open the Designer, use the configuration tools and connect to a Server running a Solution with the Online configuration checkbox enabled. The Server can be on the same computer with the configuration tools or on a remote computer.
- In this case, every change you make to the Solution is applied immediately to the running application.
- The runtime property `@Server.Redundancy.UpdateSolutionOnInactiveServer` will propagate online changes on the active server to the backup server.

Option 2: Local IP acting as the Secondary Server

- You should use the **.dbsln** file instead of **.dbrun**
- Connect the Designer on the STANDBY computer.
- Do the modifications online.
- In the application (in an administrator display), execute the command `@Server.Redundancy.SwitchToStandby()`, so the computer with the new Solution will be active.
- Finally, execute a trigger on property `@Server.Redundancy.UpdateSolutionOnInactiveServer` that will apply the changes to the other computer.
- If you want to return the Active state to the original computer, just run the `@Server.Redundancy.SwitchToStandby()` method again

Offline changes

The command line parameter /SolutionIPPathName option allows you to define a verification path. When the application starts, it verifies the Solution configuration against the reference provided on the remote computer.

For off-line changes:

- Make your Solution changes on another computer, and create the solution file to be installed for production use.
- Stop the runtime in the Standby computer, copy the solution file, and start the runtime again.

- Switch the Active to the server that is updated, and stop the runtime in the other system.
- Either use the \SolutionIPPathName parameter to enable the Standby computer to get the configuration from the server automatically when starting, or copy the file to the second computer
- Start the runtime

Hot Start

You can make changes to a Solution when it is running even if you are not connected to it. When you make changes to a Solution and apply all the new changes at once without stopping the application, it is called "hot-swapping". To do this: After you make your Solution changes, connect to the server, and click the **Hot Start** button in the **Runtime / Runtime Diagnostics** page.

Runtime changes

It is possible to update Solutions without using engineering tools. There is a method `@Server.LoadSolutionVersion(<Solution>)`, which allows dynamic loading of new Solution configuration. This command can be included with the application itself or inside an external .NET application connected to the server.

Client configuration based on Redundancy scenario

The platform's clients are easy to configure on any redundancy scenario. There is no programming or advanced configuration required. You just need to start the client station with the right parameter, as showed on **Runtime / Startup**, which builds the line in the **RichClient CmdLine** field with a **Copy** button. Example: TRichClient.exe /ip1:192.168.1.1 /port1:3101 /ip2:192.168.1.2 /port2:3101

Clients do not even need to install any Solution files on their machines. Client computers will get all the Solution information from the server. The RichClient only needs software platform on the client computer. The SmartClient is deployed through ClickOnce, so any browser with ClickOnce support can be used on the client computer.

Only the ACTIVE node serves clients. A node running as STANDBY refuses the connection, so a command line carrying a single address fails whenever that node is not the active one at that moment. Always give the client both addresses of the pair, including on a station that runs the client on one of the two servers.

This is how it works: when the client starts (either the RichClient or the SmartClient), it looks for the ip1 computer to connect. If that node does not answer, or answers as standby, the client switches to the ip2 computer. If the client loses connection with the server computer, it will try automatically to switch to the redundant pair without stopping the operator operations.

On a station where the client has to start on its own, use the AutoStartClient.exe utility rather than a shortcut in the Windows Startup folder: it polls both nodes and launches the client as soon as either one is active, instead of firing at logon before the runtime is ready. See Installation Folders and Utilities Reference.

Redundancy Interaction over Client

Clients can visualize critical redundancy information and even make a server switch using runtime properties and methods.

Switching the active server

There is a method included in the platform: `@Server.Redundancy.SwitchToStandby()`

That method will force the Active server to handle control of the Standby server; if the standby server is not running, the command fails, and the execution is kept on the same computer. If you want to use this feature, you need to create a protected display or button in an application screen.

Visualizing redundancy status

The following are the most used properties:

- `@Server.Redundancy.IsPrimary // True, if the active server is the Primary`
- `@Server.Redundancy.IsSecondary //True, if the active server is the Secondary`
- `@Server.Redundancy.IsStandByActive //True, if the standby computer is up and running`
- `@Server.Redundancy.IsSwitchToPrimaryEnabled //True, when configuration is to the Primary is always Active`
- `@Server.Redundancy.RedundancyPendingObjects //Number of objects pending synchronization`
- `@Server.Redundancy.UpdateSolutionOnInactiveServer //Allow Online changes/HotStart to replicate to the standby computer`
- `@Server.Redundancy.SwitchToStandby() // Request to the active server to switch to standby`

License Configuration for Redundancy

Each server in a redundant pair needs its own license. A license serial number activates on one computer only, under that computer's site code, so the primary and the secondary each activate their own serial. One serial cannot be activated on both servers.

License both servers with the same product and run the same license version on both, so the solution runs unchanged after a switchover. For activation, see Licensing Procedures Reference.

In this section...